KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Advanced Techniques for Grounding and Solving in the IDP Knowledge Base System

Joachim Jansen

Supervisors:
Prof. dr. ir. G. Janssens
Prof. dr. M. Denecker

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science (PhD): Computer Science

December 2016

Advanced Techniques for Grounding and Solving in the IDP Knowledge Base System

Joachim JANSEN

Examination committee:
Prof. dr. A. Bultheel, chair
Prof. dr. ir. G. Janssens, supervisor
Prof. dr. M. Denecker, supervisor
Prof. dr. B. Jacobs
Prof. dr. ir. M. Bruynooghe
Prof. dr. F. Ricca
(University of Calabria)
Docent. dr. T. Janhunen
(Aalto University School of Science)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science (PhD): Computer Science

December 2016

Preface

A chapter of over four years in my life ends with this text. During these years I have been challenged in ways I did not expect. The undoubtedly priceless upside is that this has caused me to learn about so many things and made grow in ways I had never foreseen. There were less pleasant moments (usually when I had been writing for a long time), but ultimately I have had a great time doing this PhD. Ending this chapter feels good, partly because many exciting and happy memories will remain after it has ended. There are many people that helped make these four years some of the best in my life.

First and foremost I thank my supervisors. **Gerda**, as my promotor, you have had a significant and positive impact on me and on my experience of these years. Thank you for your guidance in many things, even when they were not PhD-related. I am a better writer because you read through my writings countless times, each time providing constructive feedback. Thank you for your patience and understanding of my frustrations, even when they were not PhD-related. Thank you for all your efforts towards the quality of the education for the students at our department. We are alike in this and I do not doubt that your care for these matters has affected me. Thank you for always being there when I had one question or another. **Marc**, as my co-promotor you also had a great influence on the type of researcher I turned into. Thank you for your unceasing drive towards achieving the research goals of our group. Your countless ideas for improvement are both impressive and, at times, somewhat intimidating. I sincerely hope that, at some point in the future, you are supported by a big enough group that is able to keep up with the speed at which you create new and interesting research topics. Thank you for being critical of my work, it has helped me grow as a researcher. Thank you for the many discussions that, whilst intense and tiring, often produced excellent ideas in the end.

I am grateful to the members of my **jury**, who found time in their busy schedule to read this text and suggest many improvements. Thanks for the suggestions and for coming to Leuven for my defence.

My day-to-day experience as a PhD student was marked by the closeness and like-mindedness in our group. I believe that the way our group forms a single front on many issues is a unique asset for the future of our group. Thank you **Broes, Stef, Bart, Pieter, Jo, Ingmar, Matthias, Ruben, Laurent**, and **Tim** for the many interesting discussions, fun trips, and late-night board game sessions!

Still at the department, I enjoyed many trips to Alma and the coffee machine with **Job** and **Gijs**. You guys were a blast to talk with. Whilst our discussions were sincere and serious at heart, the topics of these discussions were often rather silly, which was exactly what I needed at times.

I also owe a big thank you to my **family**. I am, without question, a very lucky man to enjoy the support of so many people. This has never been more clear than at the times when I needed some of you most.

Last but not least, I wish to thank my wife **Steffi**. These years have been challenging, but you stood by me all this time. You listened when I complained, you motivated me when I needed it most, and you showed me how to stand tall when I wavered. I am constantly amazed by your strength and perseverance. Thank you for always being there and for making it a joy to come home after work every day. Ik hou van jou!

Abstract

The area of Knowledge Representation and Reasoning, a subfield of Artificial Intelligence, studies how knowledge can be represented and how it can be used for automated reasoning. Several declarative programming paradigms implement this by developing a formal language to symbolically represent knowledge, as well as an associated form of inference to achieve the desired solution. Recently, the Knowledge Base System (KBS) paradigm has been proposed, based on the idea that knowledge is not inherently linked to a specific reasoning task. Instead, this paradigm proposes to express knowledge in a truly declarative manner. Additionally, to stress reusability of this knowledge, the KBS paradigm allows the knowledge to be combined with one out of a set of possible inferences, each providing a solution to some type of computational task. The initial implementation of this KBS paradigm, also known as the IDP3 system, provided a suitable laboratory to examine a new type of software development.

State-of-the-art declarative programming systems such as IDP3, clasp, WASP, and lp2sat work using the *ground-and-solve* methodology. First, the high-level language is ground into a low-level propositional language. As a second step the grounding is used as input for general-purpose, low-level propositional (generally SAT-like) solvers. This thesis contains a thorough study of the impact of different grounding approaches on the solver behaviour. I.e., if the grounding process that is used is smart and results in a smaller low-level representation, does this impact the search behaviour of the underlying solver?

The language supported by the IDP3 system contains constraints and definitions. For the purpose of this thesis, we split up definitions into two kinds: *input** (also called *intentional* or *stratified* predicates, or *domain atoms*) and *search* definitions. Input* definitions are definitions that depend on concepts that are known beforehand and can be evaluated. One contribution of this thesis is the *evaluation of input* definitions using tabled Prolog.* In order to use these techniques, definitions have to be transformed into a format usable by Prolog.

The main challenge in this transformation is correctly projecting away the types and rich language constructs, whilst taking into acount Prolog's left-to-right execution mechanism. Search definitions on the other hand cannot be evaluated because they depend on unknown data. Additionally, this text describes how the above translation to Prolog can be reused to *partially evaluate* definitions, deriving as much information as possible.

We developed a method to guide the solver to focus on *relevant parts in the search space*. Only the part of the specification that is linked with the current search branch of the problem has to be taken into account. The goal of this technique is to prevent certain decisions from being made once it is possible to prove that they will not influence the outcome of searching in the current search branch. Additionally, this leads us to an *improved stopping criterion for* SAT(ID) *solvers* since any state without relevant decisions is considered an end state, instead of only states in which all literals have been decided. An implementation of this technique is presented. The implementation uses an incremental approach, adjusting certain data structures based on the changes in the solver state.

This work increases the usability of the IDP3 system and elevates it to a "mature" Knowledge Base System.

Beknopte samenvatting

Het gebied van Kennisrepresentatie en Redeneren, een onderdeel van het veld Artificiële Intelligentie, bestudeert hoe kennis kan worden voorgesteld en hoe die kan gebruikt worden voor automatische redeneertaken. Verschillende declaratieve programmeerparadigmas implementeren dit door middel van een formele taal om kennis voor te stellen en een bijbehorende vorm van inferentie om het beoogde resultaat te behalen. Recentelijk werd het Kennisbank Systeem (KBS) paradigma voorgesteld. Dit is gebaseerd op het idee dat kennis niet inherent geassocieerd is met een specifieke redeneertaak. Dit paradigma stelt voor om de kennis op een werkelijk declaratieve manier voor te stellen, in tegenstelling to eerdere aanpakken. Als een bijkomend voordeel wordt de mogelijkheid voorzien om kennis te hergebruiken voor verschillende redeneertaken die elk een oplossing bieden voor een bepaald probleem. De initiële implementatie van het KBS paradigma, ook gekend als het IDP3 systeem, heeft gediend als een geschikt laboratorium om nieuwe technieken in softwareontwikkeling te onderzoeken.

De huidige generatie aan declaratieve programmeersystemen, zoals IDP3, clasp, WASP, lp2sat, gebruiken een ground-en-solve aanpak. Bij deze aanpak wordt de specificatie met rijke taalconstructen eerst omgezet in een voorstelling die op een lager niveau werkt en geen van de rijkere taalconstructen gebruikt. Dit proces wordt *grounding* genoemd. Een tweede stap bestaat uit het verwerken van deze voorstelling met behulp van een algemene, laag-niveau solver die gewoonlijk gebaseerd is op technieken uit het SAT (satisfiability) veld. Deze thesis bevat een grondige studie over de impact van verschillende *grounding* aanpakken op het gedrag van de onderliggende solver.

De taal die ondersteund wordt door het IDP3 systeem bevat constraints en definities. In de context van deze thesis worden deze definities opgedeeld in twee categoriën: de *input** en de *zoek* definities. Input* definities zijn definities die afhangen van concepten die reeds gekend zijn, wat wil zeggen dat deze definities op voorhand uitrekenbaar zijn zonder dat er een zoekstap aan te pas komt. Dit

soort definities wordt ook wel *intentional* of *stratified* genoemd, oftwel *domain atoms* in de literatuur. Een van de bijdragen van dit werk is de evaluatie van deze *input** definities met behulp van het XSB Prolog systeem. Om deze aanpak te kunnen volgen, moeten deze definities omgezet worden naar een formaat dat bruikbaar is door het Prolog systeem. De grootste uitdagingen bij deze omzetting liggen in het correct transformeren van de types en andere rijkere taalconstructen en in het rekening houden met het uitvoeringsmechanisme van Prolog dat van links naar rechts werkt. De *zoek* definities kunnen niet op voorhand uitgerekend worden en hangen nog af van ongekende data. Deze thesis beschrijft hoe de bovenstaande omzetting kan veralgemeend worden om dit soort *zoek* definities gedeeltelijk uit te rekenen, waarbij er zoveel mogelijk informatie al wordt afgeleid.

We ontwikkelen een methode om de solver te begeleiden gedurende het zoekproces en voorrang te geven aan delen van de zoekruimte die *relevant* zijn. Enkel het deel van de zoekruimte dat verbonden is met de huidige zoektak van het probleem moet hierbij in acht genomen worden. Het doel van deze techniek is het voorkomen van beslissingen waarvan kan bewezen worden dat ze geen bijdrage hebben tot het bekomen van een oplossing in de huidige zoektak. Een tweede voordeel is dat het zoekproces een nieuw en verbeterd stopcriterium heeft, omdat de eindstatus van de solver er niet meer één hoeft te zijn waar de waarde van alle variabelen gekend is. We presenteren ook een implementatie die deze eigenschap uitbuit. Deze implementatie maakt gebruik van een incrementele aanpak, waarbij gegevensstructeren worden aangepast op basis van de veranderingen in de status van de solver.

Met dit werk wordt de bruikbaarheid van het IDP3 systeem verbeterd en kan het systeem zich profileren als een (steeds) robuustere vorm van deze alternatieve aanpak tot softwareontwikkeling.

Contents

List of Acronyms

ASP Answer Set Programming

BDD Binary Decision Diagram

CDCL Conflict-Driven Clause Learning

CNF Conjunctive Normal Form

CP Constraint Programming

CSP Constraint Satisfaction Problem

DEFNF Definition Normal Form

ECNF Extended Conjunctive Normal Form

FO First-Order Logic

IDP Imperative Declarative Programming

KBS Knowledge Base System

KR Knowledge Representation

LP Logic Programming

SMT SAT Modulo Theories

VSIDS Variable State Independent Decaying Sum

WFS Well-Founded Semantics

List of Symbols

$\mathcal{T}^{\phi}$ the set of formulas in a theory

$F(\bar{t})$ a function term

Σ^{F} the set of function symbols in a vocabulary Σ

$A \cap B$ the intersection of sets A and B

$A \cup B$ the union of sets A and B

$A \subset B$ A is a strict subset of B

$A \subseteq B$ A is a subset of B

E a set expression

F a function symbol

I an interpretation, also called a structure

L a list

P a predicate symbol

R the set of rules in a definition

$\leq_p$ precision relation on interpretations

$\leq_t$ truthness relation on truth values

Δ a definition

Σ a vocabulary

Υ a typing

$\mathtt{agg}_f$ an aggregate function

$\texttt{agg}_t$ an aggregate term

$\bar{t}$ a list of terms $[t_1, \ldots, t_n]$

$\bar{v}$ a list of variables $[v_1, \ldots, v_n]$

$\bar{a}$ a tuple

$\bigodot$ an FO$(\cdot)$ built-in arithmetic operator

F^{-1} inverse function of bijection F

$L[i]$ accessing the i-th element of list L

$\mathbb{H}$ the type containing all symbols values

$\mathbb{I}$ the type containing all integers

$\mathbb{N}$ the type containing all natural numbers

$\mathcal{D}$ the set of definitions in a theory

$\mathcal{I}$ a partial (three-valued) interpretation, also called a partial structure

$\mathcal{P}(S)$ powerset of set S

$\mathcal{T}$ a logical theory

ϕ a formula

$\sim$ an FO$(\cdot)$ built-in term comparison

d a domain element

l a literal

t a term

val an FO$(\cdot)$ built-in term value

v a variable

Σ^P the set of predicate symbols in a vocabulary Σ

$\mathbb{B}$ the type containing all Boolean values $\{\texttt{true}, \texttt{false}, \texttt{unknown}\}$

`card` aggregate function that takes the number of elements in a set

`max` aggregate function that takes the maximum of a set

`min` aggregate function that takes the minimum of a set

`prod` aggregate function that takes the product of all elements in a set

`sum` aggregate function that takes the sum of all elements in a set

$\Sigma^{\mathbb{T}}$ the set of type symbols in vocabulary Σ

$\mathbb{T}$ a type symbol

`P` a Prolog program

List of Figures

List of Tables

Chapter 1

Introduction

The world has embraced computers in a way that was completely unforeseen when the first computers and computer networks were designed. This has resulted in a near-instant global communication network and everyday computing devices that dwarf the machines that were used for the space exploration half a century ago. With this infrastructure, it has never before been easier to access information and learn new things. The last generation in the western world to know what it is like not to be able to instantly look up *anything* is alive today. That is, if you bar the coming of a new dark age. Due to wealth disparity most of what was mentioned above is not yet available to all humans. However, there are initiatives [48, 79] that aim to improve infrastructure to make this a global phenomenon. Recent reports [77, 84] indicate that this would not only lead to an increased connectivity, but also to a significant boost in economic growth.

This change also holds for academics: in Europe, more people than ever are enjoying a higher education [50]. The European Union has adopted a policy to stimulate higher education to the point where 40% of those aged 30-34 have had such education by 2020 [37]. Thanks to this global communication it has also become easier to set up international scientific cooperation and communication.

This drastic increase in computing power also means that there are many tasks that are (able to be) solved by machines. This is governed by the field of Computer Science (CS). Artificial Intelligence (AI), a subfield of CS, aims to provide a machine with human-like intelligence so that they may solve some of the problems that humans encounter. The presented research in this thesis is a small step in this process. To understand our perspective, first consider that specific *programs* must be created in order to allow a machine

to perform certain tasks and solve certain problems. The creation of such programs is called *software development*. My aim is to further enhance an alternative software development method. In classical ("imperative") software development approaches the knowledge surrounding the problem that is to be solved (henceforth called the "domain knowledge") is embedded into the data structures and the procedural flow of the software. We consider this a weakness for multiple reasons.

- An outside programmer that is new to a software project may experience great difficulty getting familiarized with the code base of the project. This is especially the case if the software project is big and contains a large amount or complex domain knowledge. There is a great dependency on the provided documentation.
- It is easier to introduce faulty code for complex domain knowledge if the form in which it has to be written down is convoluted and not at all similar to the way the programmer understands the domain.
- If there is a change in the domain knowledge, it requires a programmer that is experienced in the software project in question to quickly implement the change into the code base. Outside programmers may find it hard to correctly identify where the change in code must be implemented.
- This approach is called "imperative" because the programmer specifies to the machine, using commands, *how* the problem should be solved. A weakness therefore is that, for some problems, it is very difficult to come up with a series of commands that solve the problem. A well-known example of this is a *sudoku* puzzle. It is easy to explain to someone what the rules of a sudoku puzzle are. It is, however, far more difficult to express *how* such a puzzle must be solved.

There is an alternative to this "imperative" approach called the "declarative" programming paradigm. For this paradigm the focus lies in informing the machine *what* the problem is, and to rely on general-purpose algorithms for solving the problem. The specific instance of declarative programming that is considered in this text is the Knowledge Base System (KBS) paradigm. It focuses on clearly representing the underlying problem that is to be solved. In addition to this, it is designed to facilitate reuse of this representation to solve different tasks associated with the same domain knowledge. The KBS paradigm allows explicit representation of this knowledge in an expressive, intuitive, and formal language using a process called "modeling". This paradigm relies on techniques and practices from the field of Knowledge Representation (KR): the study of how knowledge can be represented and how it can be used for automated, machine-driven, problem solving.

The language that is used in this text is an extension of First-Order Logic (FO), formal language to perform inferences. However, there several shortcomings have been identified to its form. In order to solve these shortcomings, several additions to the language have been made over the years. Such additions are, for example **(1)** the usage of *types*, **(2)** rule-based reasoning to express concepts formerly not expressible in FO (such as the transitive closure of a binary relation) as part of a *definition* for that concept, **(3)** function symbols that map to a value, and **(4)** expressions concerning aggregate functions on complex *set expressions*. We use FO($\cdot$) to indicate a family of languages that extend FO. In the remainder of this text, we abuse notation and talk about the language FO($\cdot$) as the specific instance of the FO($\cdot$) family that contains all the extensions discussed in this thesis.

The IDP3 system was developed to support the FO($\cdot$) language and allow end-users to develop software according to the KBS paradigm. The IDP3 system is a ground-and-solve system. This means that, the system works in two phases: during the *grounding* phase, the input specifications (in FO($\cdot$)) are transformed into a lower-level representation. This lower-level presentation is generally *quantifier-free* and is fed into a state-of-the-art solver as part of the *solving* phase. The solver that is used in the IDP3 system is called MiniSAT(ID) [21, 65].

It is the goal of the research presented in this thesis to better understand and extend IDP3. As such, each chapter in this thesis focuses on a specific area within IDP3 that is investigated or extended in some way. We intend to, by the end of this text, have gained sufficient understanding of and provided enough extensions to the IDP3 system to have it be considered a new, more "mature" system that has an increased robustness and functionality.

Contributions

The main contributions of the presented research are:

- **Chapter 4** contains two experiments: **(1)** a rigorous analysis of the performance of different combinations of *grounding* techniques in the IDP3 system and **(2)** an investigation of the impact of the usage of these techniques on the subsequent *solving* phase. The second experiment leads to the conclusion that a larger grounding (that could have been smaller) causes a larger overhead during the solving phase. However, no statistically significant correlation between an increased grounding size and an increased number of conflicts was found, which means that more naive, larger, groundings, do not lead to a more complex solving process

with a bigger search tree. This work is published in the proceedings of the 16th "International Symposium on Principles and Practice of Declarative Programming" (PPDP'14) [53].

- **Chapter 5** identifies a subtask, called *evaluating input∗ symbols*, in the workflow of IDP3 for which no search is needed and introduces a transformation of that subtask to a different formalism. As a result, a system (`XSB`) supporting that formalism (Tabled Prolog) can be used to perform this task for IDP3 instead of writing a dedicated algorithm for it. The addition of this transformation causes essential speedups for IDP3 when dealing with problems that rely on the evaluation of input∗ symbols and puts the performance of IDP3 in the same order of magnitude with its main competitors. This work is published in the journal "Theory and Practice of Logic Programming", volume 13, issue 4-5 [56], and in the proceedings of the 14th Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'14) [55].
- **Chapter 6** identifies a formal property, called *relevance*, for SAT(ID) solvers which identifies propositional variables that are useless (called *irrelevant*) for further search. This chapter suggests modifications to existing SAT(ID) solvers to exploit these properties and gives detailed instructions on how to implement the suggested modifications in an existing SAT(ID) solver. Variable State Independent Decaying Sum (VSIDS), considered the best general-purpose SAT heuristic, was observed to make a significant amount of *irrelevant* decisions in an experimental evaluation. This work is published in the proceedings of the 25th "International Joint Conference on Artificial Intelligence" (IJCAI'16) [52] and the proceedings of the 9th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'16) [54].

Structure of the Text

The rest of the thesis is structured as follows.

- **Chapters 2 and 3** serve as an introduction for understanding and developing a KBS based on the FO(·) language. Chapter 2 discusses core mathematical concepts (Section 2.1), the syntax of FO(·) (Section 2.2), and semantics of FO(·) (Section 2.3). Chapter 3 gives a short, high-level introduction to KBS paradigm (Section 3.1) and the IDP3 system (Section 3.2). Of special interest is the section that gives a high-level overview on the existing techniques for grounding FO(·) specifications (Section 3.2.1).

- **Chapter 4** contains two experiments that investigate the grounding phase. Section 4.1 presents the experiment where the performance is measured in several ways for different combinations of grounding techniques. This results in a thorough and highly practical cost/benefit analysis of these grounding techniques. In Section 4.2, we research how an increase in the grounding size (by grounding naively) impacts the performance of the subsequent solving phase. The result of this is that an increase in grounding size may lead to an increase in the overhead during solving, but does not lead to a bigger search tree.

- **Chapter 5** introduces a way of solving a deterministic subtask in IDP3 without performing search. Section 5.1 presents a more complex version of the workflow of IDP3 and identifies a subtask that does definition evaluation. A translation of the input of this definition evaluation step into a Logic Programming (LP) problem is described (Section 5.2.1). We then show how, using the `XSB` tabled Prolog system, this translation can be used to perform definition evaluation (Sections 5.2.2 and 5.2.3). Experiments show that this addition puts us on even foot compared to other state-of-the-art ground-and-solve systems for problems that rely on definition evaluation (Section 5.3). As an extension to this, we also show how to *refine* a more general class of definitions (Section 5.4).

- **Chapter 6** introduces an extension to the solver of IDP3, starting with a description of justifications and Definition Normal Form (DEFNF) specifications (Section 6.2). A formal property (*relevance*) of literals in the solving process is identified, which allows us to prove that literals that do not have this property cannot possibly contribute to the search process for solutions that would originate from the current partial assignment in the solver (Section 6.3). Next, detailed instructions on how to implement the suggested modifications in an existing SAT(ID) solver are presented (Section 6.4). Experiments show that exploiting this property enables the solver to prevent a significant amount of decisions. However, they also show that the current implementation still suffers from a substantial overhead cost (Section 6.5).

Chapter 2

Preliminaries: $\mathrm{FO}(\cdot)$

This chapter and the next one serve as a summary of all the background knowledge that is assumed in the remainder of this text. Section 2.1 lists the mathematical constructs and notations that are needed. In Section 2.2, we present the language FO(·) that is used to represent knowledge. Section 2.3 discusses semantics that are associated with languages such as FO(·).

2.1 Some Mathematical Basics

In this section we establish the mathematical constructs used. We use set theory, ordered lists, typings, and tuples. We call the items in a set, list, or tuple *elements*.

2.1.1 Sets

Unless mentioned otherwise, sets cannot contain multiple instances of the same element. I.e., the sets are not multisets.

Given sets A, B, C and elements a, $a_1, \ldots, a_n$:

Representation : We use $\{a_1, \ldots, a_n\}$ to represent the set that contains n elements, namely a_1, a_2, ..., a_n.

Element of : $a \in A$ indicates that a is in A.

Empty set : $\emptyset$ indicates the empty set (that contains no elements).

Set size : $|A| = n$ indicates that A contains n elements.

Set equality : $A = B$ indicates that A contains the same elements as B. We use $A \neq B$ to indicate that $A = B$ does not hold.

Union : $A \cup B = C$ indicates that C contains exactly those elements that are either in A or in B.

Subset of : $A \subseteq B$ indicates that $A \cup B = B$.

Strict subset of : $A \subset B$ indicates that $A \subseteq B$ and $A \neq B$.

Intersection : $A \cap B = C$ indicates that C contains exactly those elements that are both in A and in B.

Minus : $A \setminus B = C$ indicates that C contains exactly those elements that are in A and *not* in B.

Powerset : $\mathcal{P}(A) = B$ indicates that B is the set of all subsets of A (including the empty set).

Theoretically, a set can be an element of another set (resp. list or tuple). In this section, we use the term *value* to express that something is a non-set. Unless specified otherwise, the assumption is made that an element denotes a value.

Given a (partial/total) ordering $\leq$ over elements, we use $A \leq B$ to indicate that for all $a \in A$ and $b \in B$ holds $a \leq b$.

2.1.2 Lists

A list (L) is an ordered collection of elements. Lists can contain elements multiple times at different indices. We use $(a_1, \ldots, a_n)$ to represent a list of length n that contains elements a_1 through a_n, in that order. The first item in the list has index 1, the second item in the list has index 2 etc. If a is an element of L with index i, we also call a the i-th element in L.

Given some list L and an element a:

Empty list : $\emptyset$ indicates the empty list (that contains no elements).

List size : $|L| = n$ indicates that L is a list of length n.

Element access : $a = L[i]$ indicates that a is the i-th element in L.

2.1.3 Type

A type is a mathematical concept to represent a set of values. A type is also associated with a *type symbol* (usually denoted as $\mathbb{T}$).

2.1.4 Typings and Tuples

A typing (Υ) is a list of types. We use the notation $\langle \mathbb{T}_1, \ldots, \mathbb{T}_n \rangle$ to indicate a typing of length n that contains the types $\mathbb{T}_1$ through $\mathbb{T}_n$, in that order. We reuse the list size and element access notation. I.e., $|\Upsilon| = n$ indicates that Υ is of length n and $\Upsilon[i] = S_i$ indicates the i-th type of a typing.

A list L *respects* typing Υ if

(1) $|L| = |\Upsilon|$ and

(2) for $1 \leq i \leq n$, $L[i] \in \Upsilon[i]$.

I.e., if a list L respects some typing Υ, it represents a list of values coming from the corresponding list of types of Υ. We use the term *tuple* to denote such lists. Thus, tuple $(\bar{a})$ is a list that is usually associated with some typing.

We use the notation $(a_1, \ldots, a_n)$ to represent the tuple that contains elements a_1 through a_n. In this text, often other lowercase letters $(\bar{i}, \bar{o}, \bar{t})$ may be used to represent tuples. We reuse the list size and element access notation. I.e., $|\bar{a}| = n$ indicates that tuple $\bar{a}$ is of length n and $\bar{a}[i] = a_i$ indicates the i-th element of a tuple.

We use $\bar{a} \in \Upsilon$ to indicate that tuple $\bar{a}$ respects typing Υ. We also use the term *n-tuple* to refer to a tuple of length n.

2.1.5 Functions

A function is a tool for mapping *input tuples* to *output tuples*. It is associated with two typings: an *input typing* and an *output typing*. Input (resp. output) tuples must respect the input (resp. output) typing. A function is also associated with a *function symbol* (usually denoted as F). If a function has input typing $\langle I_1, \ldots, I_n >$ and output typing $< O_1, \ldots, O_m \rangle$, the *signature* of that function is $F : I_1 \times \cdots \times I_n \rightarrow O_1 \times \cdots \times O_m$. The length of the input typing is called the *arity* of the function. I.e., the function above has arity n. We also use F/n to refer to this function.

A function maps each input tuple to exactly one output tuple. If a function maps the input tuple $(i_1, \ldots, i_n)$ to output tuple $(o_1, \ldots, o_m)$ this is denoted using $F(i_1, \ldots, i_n) = (o_1, \ldots, o_m)$. If the output typing has length one, we also use $F(i_1, \ldots, i_n) = o_1$.

A function is *injective* if every possible output tuple is mapped to by *at most* one input tuple. An injective function is called an *injection*. A function is *surjective* if every possible output tuple is mapped to by *at least* one input tuple. A surjective function is called a *surjection*. A function is *bijective* if it is injective as well as surjective. A bijective function is called a *bijection*.

A bijection has an *inverse* function, with function symbol F^{-1} that maps output tuples of the original function to input tuples of the original function. I.e.,

$$F^{-1}(F(i_1, \ldots, i_n)) = (i_1, \ldots, i_n)$$

where $(i_1, \ldots, i_n) \in \langle I_1, \ldots, I_n \rangle$ is any input tuple for F.

We call a function *partial* if it does not provide a mapping for each of its possible input tuples. A function that is non-partial is called a *total* function. Whenever we use the term "function" without specifying whether it is total or not, we mean a total function.

In the remainder of this text we often indicate a function by referring to its function symbol. I.e., saying "function F maps ..." means that we are talking about a function with function symbol F and use F to refer to that function.

2.1.6 Predicate

A predicate is a (total) function that maps to the typing $\langle \mathbb{B} \rangle$, with $\mathbb{B} = \{\texttt{true}, \texttt{unknown}, \texttt{false}\}$ the set of Boolean values. A predicate is also associated with a *predicate symbol* (usually denoted by P).

2.2 The Language FO(·)

In this section we introduce the logical constructs that are part of the FO(·) language and the role they fullfil with respect to representing knowledge. FO(·) is an extension of First-Order Logic (FO) with **(1)** inductive definitions, **(2)** (partial) functions, **(3)** types, and **(4)** aggregates. An FO(·) encoding consists of three top-level constructs: the *vocabulary*, the *interpretation*, and the *theory*. These constructs are further built upon the basic mathematical constructs provided in Section 2.1.

```
vocabulary Σ1 {
    type 𝕋
    type subT  isa 𝕋
}
```

Listing 2.2: An example notation of a vocabulary with some types

The goal of this language is to provide an end-user with an intuitive to specify knowledge in a formal manner. To this end, the FO(·) language has had inspiration for its extensions of FO come from different areas. The concept of inductive definitions originates in classical mathematics, functions and aggregates are also present in, for example, Constraint Programming (CP), and several type systems [49, 58, 72] have been already suggested for formal languages.

2.2.1 Vocabulary Σ

A vocabulary represents the ontology (i.e., the symbols that are used) of the logical specification. These are either type symbols, predicate symbols, or function symbols. A vocabulary is usually denoted with Σ. We use $\Sigma^{\mathbb{T}}$ (resp. Σ^{P},Σ^{F}) to denote the set of type symbols (resp. predicate symbols, function symbols) in Σ. We use $\Sigma = \langle \Sigma^{\mathbb{T}}, \Sigma^{P}, \Sigma^{F} \rangle$ to denote these three sets in the vocabulary.

We use the following notation to more easily represent the contents of a vocabulary Σ. Note that we use `//` to indicate commented lines.

```
vocabulary Σ  {
    //some vocabulary content
}
```

Listing 2.1: An example notation of a vocabulary

The vocabulary contains type symbols of the types that will be used in the FO(·) specification. Informally, a type with type symbol $\mathbb{T}$ is a set denoting a certain "type" of elements. A vocabulary cannot contain duplicate type symbols. A type can be a subtype of any other type. We use the keyword `isa` to indicate this subtype relation.

Example 2.2.1. The vocabulary shown in Listing 2.2 has two types: $\mathbb{T}$ and *sub*$\mathbb{T}$. In addition to this, *sub*$\mathbb{T}$ is a subtype of $\mathbb{T}$.

The vocabulary contains function symbols of the functions that will be used in the FO(·) specification. Informally, a function with function symbol F is

used to map input tuples to elements. As opposed to the formal definition of a function in Section 2.1.5, we impose that, for FO(·), functions in vocabularies have an output typing of length 1. The output tuple of length one is then also called the output value. We use $F(I_1, \ldots, I_n) : O$ as a shorthand to indicate a function with function symbol F, input typing $\langle I_1, \ldots, I_n \rangle$, and output typing $\langle O \rangle$. Readers must note that these functions are *not* Herbrand functions, since they map to a pre-existing type.

Example 2.2.2. `birthYearOf(Person):Year` is a function with function symbol `birthYearOf`, arity 1, and typing ⟨`Person,Year`⟩. It could be used to represent the year in which people have been born. `birthMonthOf(Person):Year` is a function with function symbol `birthMonthOf`, arity 1, and typing ⟨`Person,Month`⟩. It could be used to represent the month in which people have been born. The more fluent representation is shown below.

```
vocabulary Σ2 {
   type Person
   type Year
   type Month
   birthYearOf(Person):Year
   birthMonthOf(Person):Month
}
```

Listing 2.3: An example notation of a vocabulary with some functions

The vocabulary contains predicate symbols of the predicates that will be used in the FO(·) specification. Informally, a predicate with predicate symbol (P) is used to represent relations between elements. I.e., if a predicate maps tuple (a_1, a_2) to `true`, that means that the tuple is in the relation. We use $P(I_1, \ldots, I_n)$ as a shorthand to indicate a predicate with predicate symbol P and input typing $\langle I_1, \ldots, I_n \rangle$. The output typing of $\langle \mathbb{B} \rangle$ is left implicit. In order to disambiguate between predicates and functions, we impose that Σ contains no duplicate predicate nor function symbols.

Example 2.2.3. `inEuropeanUnion(Country)` is a predicate that has name `inEuropeanUnion`, arity 1, and typing ⟨`Country`⟩. It could be used to represent which countries are in the European Union. The more fluent representation is shown below.

```
vocabulary Σ3 {
   type Country
   inEuropeanUnion(Country)
}
```

Listing 2.4: An example notation of a vocabulary with a predicate

```
structure I : Σ {
  // some type, predicate, and function interpretations
}
```

Listing 2.5: An example notation of a structure I over vocabulary Σ

For ease of notation, we extend some of the set operations to vocabularies. We take the union of two vocabularies by taking the union of their type symbols, predicate symbols, and function symbols. Hence, $\Sigma_1 \cup \Sigma_2 = \langle \Sigma_1^{\mathbb{T}} \cup \Sigma_2^{\mathbb{T}}, \Sigma_1^P \cup \Sigma_2^P, \Sigma_1^F \cup \Sigma_2^F \rangle$. We define the intersection ($\Sigma_1 \cap \Sigma_2$) and minus ($\Sigma_1 \setminus \Sigma_2$) of vocabularies in the same manner. We sometimes abuse notation and speak of the types (resp. functions, predicates) present in a vocabulary to denote the type symbols (resp. function symbols, predicate symbols) in that vocabulary.

Every vocabulary implicitly contains a number of built-in type symbols, predicate symbols and function symbols. These are never explicitly shown. The built-in type symbols are $\mathbb{B}$ (the set of Boolean values $\{\texttt{true}, \texttt{false}, \texttt{unknown}\}$, $\mathbb{I}$ (the set of integers) and $\mathbb{N}$ (the set of natural numbers), a subtype of $\mathbb{I}$.

The built-in predicate symbols are the relational operators ($=$, $<$, $>$, $\leq$, $\geq$) for which we support infix notation. The built-in function symbols are binary arithmetic operators: addition ($+$), subtraction ($-$), multiplication ($\times$), integer division ($/$), and modulo ($\%$) for which we support infix notation.

We use $\sim$, resp. $\odot$, to represent any of the built-in relational operators, resp. built-in arithmetic (partial) functions. A built-in relational operator $\sim$ can be applied to any two elements. A built-in arithmetic operator $\odot$ can be applied to any two *integers*. The meaning of these built-in constructs is discussed in the next section.

2.2.2 Interpretation I

Given a vocabulary Σ, an *interpretation* (I) over Σ is used to associate a meaning to the symbols in the vocabulary. We use $\mathbb{T}^I$ (P^I, F^I) to denote the interpretation of type $\mathbb{T}$ (resp. predicate P, function F). We use the following notation to more easily represent the contents of a structure I over vocabulary Σ. Because we use the keyword `structure` for this kind of notation, we use the terms "interpretation" and "structure" interchangeably.

The basic concept used in a structure is a *domain element*. Domain elements can be Boolean values ($\mathbb{B}$), integers ($\mathbb{I}$), or symbols ($\mathbb{H}$). The interpretation of a type is the set of domain elements that the type contains. A *valid* interpretation

```
structure I1 : Σ1 {
    type 𝕋 = {Triangle; 23; true }
}
```

Listing 2.6: An example notation of an (invalid) interpretation of Σ_1

is one that is well-formed. In order for an interpretation to be *valid* it must contain a type interpretation for all types in its associated vocabulary. In the next sections we define additional *validity constraints* on other constructs of FO(·) encodings. Unless specified otherwise, we always assume that a given FO(·) construct is valid.

Example 2.2.4. Given the vocabulary in Example 2.2.1, the type interpretation $\mathbb{T}^I = \{Triangle, 23, \texttt{true}\}$ indicates that the type with type symbol $\mathbb{T}$ contains exactly the domain elements **(1)** the Boolean value `true`, **(2)** the integer 23, and **(3)** the symbol *Triangle*. This is shown in a more convenient notation in Listing 2.6. Note that this interpretation I_1 is not valid because it does not contain a type interpretation for $sub\mathbb{T}$.

The interpretation of a function $F(I_1, \ldots, I_n) : O$ is a specification of which input tuples map to which output values. We represent this using the function with signature $F^I : \langle I_1, \ldots, I_n, O\rangle \to \mathbb{B}$. We use Boolean values to indicate whether a mapping is present. $F^I((in_1, \ldots, in_n, out)) = \texttt{true}$ (resp. `false`, `unknown`) indicates that the mapping from $(in_1, \ldots, in_n)$ to the value *out* is present in the function (resp. not present, may or may not be present). In order for the interpretation of a (partial or non-partial) function to be valid, F^I must stipulate that any input tuple maps to at most one value. In addition to this, for the interpretation of a non-partial function to be valid, F^I cannot stipulate that there is an input tuple that does not map to any value. Note that in addition to this, F^I must always map each input tuple of F^I to either `true`, `false`, or `unknown`. In order for an interpretation to be valid all its function interpretations must be valid.

Example 2.2.5. Given Σ_2 from Example 2.2.2, the following are some possible interpretations. $\texttt{birthYearOf}^I(\texttt{Joe}, 1996) = \texttt{true}$ indicates that the function `birthYearOf` must map `Joe` to the number 1996, with the intended meaning that `Joe` was born in 1996. $\texttt{birthYearOf}^I(\texttt{Joe}, 1997) = \texttt{false}$ indicates that the function `birthYearOf` must **not** map `Joe` to the number 1997, with the intended meaning that it is known that `Joe` was **not** born in 1997. $\texttt{birthMonthOf}^I(\texttt{Joe}, 1) = \texttt{unknown}$ indicate that the function `birthMonthOf` may or may not map `Joe` to the number 1, with the intended meaning that it is not known whether or not `Joe` was born in January. Similarly, $\texttt{birthMonthOf}^I(\texttt{Joe}, 2) = \texttt{unknown}$ indicates that it is not known whether `Joe` was born in February. A more fluent

```
structure I2 : Σ2 {
  type Person = {Joe}
  type Year = {1996..1998}
  type Month = {1..12}
  birthYearOf<ct> = {Joe->1996}
  birthYearOf<cf> = {Joe->1997}
  birthMonthOf<ct> = {}
  birthMonthOf<u> = {Joe->1; Joe->2}
  // the next line is implied by previous two lines
  // birthMonthOf<cf> = {Joe->3; ...; Joe->12}
}
```

Listing 2.7: An example notation of an interpretation of Σ_2

representation of an extension of the above examples is shown in Listing 2.7. In this representation we also implicitly express that `Joe` was not born in months 3 through 12 by stating that there is no month of which we are certain that `Joe` was born in it. This is because (`Joe`,1) and (`Joe`,2) are the only tuples that map to `unknown`. Recall that we imposed that each input tuple to $\texttt{birthMonthOf}^I$ (i.e., each `Person-Month` combination) must map to exactly one Boolean value. If there are no tuples that map to `true`, it must be that the remaining tuples map to `false` for the interpretation to be valid. In the same manner, whilst $\texttt{birthYearOf}^I(\texttt{Joe}, 1998) = \texttt{unknown}$ in Listing 2.7, one can derive that $\texttt{birthYearOf}^I(\texttt{Joe}, 1998) = \texttt{false}$ in any valid interpretation, because it $\texttt{birthYearOf}^I(\texttt{Joe}, 1996) = \texttt{true}$.

The interpretation of a predicate P/n is a specification of which n-tuples are in the relation represented by P and which n-tuples are not. We represent this using a (meta)function P^I with signature $P^I : \Upsilon \to \mathbb{B}$ where Υ is the input typing of P (note that the Boolean target type is not present in the typing of a predicate).

Example 2.2.6. Given Σ_3 from Example 2.2.3, the following are some possible interpretations. $\texttt{inEuropeanUnion}^I(\texttt{Belgium}) = \texttt{true}$ indicates that the tuple (`Belgium`) is in the relation `inEuropeanUnion`, i.e., that Belgium is in the EU. $\texttt{inEuropeanUnion}^I(\texttt{China}) = \texttt{false}$ indicates that the tuple (`China`) is **not** in the relation `inEuropeanUnion`, i.e., that China is **not** in the EU. $\texttt{inEuropeanUnion}^I(\texttt{UnitedKingdom}) = \texttt{unknown}$ indicates that the tuple (`UnitedKingdom`) may or may not be in the relation `inEuropeanUnion`, i.e., that it is not known whether the United Kingdom is in the EU. The more fluent representation is shown in Listing 2.8.

Definition 2.2.7 (Two-valued, three-valued interpretation)**.** If the interpretation of a function or predicate does not map any tuple to `unknown`, then

```
structure I3 : Σ3 {
   type Country = {Belgium;France;China;UnitedKingdom}
   inEuropeanUnion<ct> = {Belgium;France}
   inEuropeanUnion<cf> = {China}
   inEuropeanUnion<u>  = {UnitedKingdom}
}
```

Listing 2.8: An example notation of an interpretation of Σ_3

we say that the interpretation for that function or predicate is *two-valued*. If interpretation I has only two-valued function and/or predicate interpretations, we call it a *two-valued* interpretation. Otherwise, we call I a *partial* or *three-valued* interpretation.

This comes from the fact that two-valued function/predicate interpretations only map to two values: {`true`, `false`}. If we speak of an interpretation without mentioning whether it is two-valued or not, we intend to mean a three-valued interpretation.

Definition 2.2.8 (Projection)**.** Given an interpretation I and a vocabulary Σ, we define the *projection* of I onto a sub-vocabulary Σ' of Σ (denoted $I\,|_{\Sigma'}$) as the interpretation that contains only interpretations of types, predicates, and functions that are in Σ'.

Definition 2.2.9 (Precision order on structures)**.** We define the *precision order* on truth domain elements, denoted $<_p$, as follows: `unknown` $<_p$ `true`, `unknown` $<_p$ `false`. We extend the precision relation on truth domain elements to a precision relation on interpretations. We define the precision relation on interpretations as follows: $I \leq_p I'$ if and only if both interpretations have the same vocabulary (Σ) and

- $\mathbb{T}^I = \mathbb{T}^{I'}$ for every type $\mathbb{T} \in \Sigma^{\mathbb{T}}$,
- for every predicate $P(\mathbb{T}_1, \ldots, \mathbb{T}_n) \in \Sigma^P$ and for every $(t_1, \ldots, t_n) \in \langle \mathbb{T}_1, \ldots, \mathbb{T}_n \rangle$, $P^I(t_1, \ldots, t_n) \leq_p P^{I'}(t_1, \ldots, t_n)$, and
- for every function $F(\mathbb{T}_1, \ldots, \mathbb{T}_n) : \mathbb{T}_o \in \Sigma^F$ and for every $(t_1, \ldots, t_n, t_{out}) \in \langle \mathbb{T}_1, \ldots, \mathbb{T}_n, \mathbb{T}_o \rangle$, $F^I(t_1, \ldots, t_n, t_{out}) \leq_p F^{I'}(t_1, \ldots, t_n, t_{out})$.

The interpretation of relational operators is built-in and works for any domain element. We use the classical term ordering as in Prolog.

Example 2.2.10. Some examples of this are (with 1,3 $\in \mathbb{I}$ and Belgium, China, France $\in \mathbb{H}$)

- $1 \leq 3$,
- $1 < 3$,
- Belgium $<$ China, and
- China $<$ France.

The interpretation of arithmetic operators are built-in for integers (note that, in this way, only the integer division is provided).

Example 2.2.11. Some examples of this are

- $4 = 1 + 3$,
- $3 = 1 \times 3$, and
- $3 = 3 \ / \ 1$.

2.2.3 Theory $\mathcal{T}$

Given a vocabulary Σ, a *theory* ($\mathcal{T}$) over Σ is used to express the desired properties of and restrictions on the symbols in Σ. In this section we give a formal description of the components of a theory.

A theory contains a set of *formulas* and a set of *definitions*. The set of formulas (resp. definitions) in a theory is denoted using $\mathcal{T}^{\phi}$ (resp $\mathcal{T}^{\Delta}$).

We use the following notation to represent the contents of a theory $\mathcal{T}$ over vocabulary Σ.

```
vocabulary T : Σ {
  {
    //some definition
  }

  {
    //another definition
  }

  // some formulas
}
```

Listing 2.9: An example notation of a theory $\mathcal{T}$ over vocabulary Σ

We now start a bottom-up description of all concepts that can be present in a theory.

Terms

A term (t) can be one of the following: **(1)** a built-in value (val), **(2)** a variable (v), **(3)** a function application term ($F(\bar{t})$), **(4)** or an aggregate term ($\texttt{agg}_t$). In the text below each of these kinds of terms is defined. We use $\bar{t}$ to denote a sequence of terms. Every term is also associated with a *type*.

In the scope of this thesis, a built-in value term (val) can only be an integer value. The type of an integer value is the integer type ($\mathbb{I}$)

A variable (v) is a typed symbolic placeholder for another term of that type. We use $v[\mathbb{T}]$ to indicate that v is a variable of type $\mathbb{T}$. We use $\bar{v}$ to denote a sequence of variables. Since variables are placeholders, they can be *bound* to other values. If a variable v is not bound in an expression, we call that variable *free* in that expression. Theory constructs that will be introduced later can provide bindings. With c being some other theory construct, we use **(1)** $c[\bar{v}]$ to indicate that every $v \in \bar{v}$ is free in c and **(2)** $c\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ to indicate that variable v_i is bound to term t_i in c. This means that when evaluating the term v_i, the value of t_i is used instead.

A function application term ($F(\bar{t})$) is the application of a function symbol (F) to a tuple of terms ($\bar{t}$), called its arguments. In order for a function application term to be *valid*, the list of arguments has to respect the input typing of F.

An aggregate term ($\texttt{agg}_t$) is an aggregate function ($\texttt{agg}_f$) applied to a setexpression (E). We postpone the definition of a set expression until some other theory constructs are defined (see Section 2.2.3)

Predicate Application

A predicate application ($P(t_1, \ldots, t_n)$) is the application of a predicate symbol from Σ^P to a tuple of terms. In order for a predicate application to be *valid*, it must **(1)** be a predicate with a *valid* name, **(2)** have an argument tuple of the correct size, **(3)** have an argument tuple of the correct typing, and **(4)** contain no free variables. A predicate has a valid name if it is one of the build-in relational operators, or if its name is any combination of lower- and uppercase letters. A predicate application of a built-in relational operator presents itself in infix notation ($t_1 \sim t_2$).

Formulas

A formula (ϕ) can be a predicate application, a negation, a conjunction, a disjunction, a universally quantified formula, or an existentially quantified formula.

Negation: $\neg\phi$ A negation has a single subformula. A negated formula is satisfied if and only if its subformula is not satisfied.

Conjunction: $\phi_1 \wedge \cdots \wedge \phi_n$. A conjunction has subformulas and indicates that in order for the conjunction to be satisfied, all of its subformulas have to be satisfied.

Disjunction: $\phi_1 \vee \cdots \vee \phi_n$. A disjunction has subformulas and indicates that in order for the disjunction to be satisfied, at least one of its subformulas has to be satisfied.

Universally quantified Formulas: $\forall v : \phi_1$. A universally quantified formula has a quantified variable over some type ($v[\mathbb{T}]$) and a subformula (ϕ_1). A universally quantified formula is satisfied if and only if for all domain elements d in the type $\mathbb{T}$, $\phi_1\{v \mapsto d\}$ is satisfied. We also say that a universally quantified formula *binds* the variable v. We use

$$\forall v_1 \dots v_n : \phi_1.$$

or

$$\forall \bar{v} : \phi_1.$$

as a shorthand for

$$\forall v_1 : (\forall v_2 : (\dots (\forall v_n : \phi_1) \dots)).$$

Existentially quantified Formulas: $\exists v : \phi_1$. An existentially quantified formula has a quantified variable over some type ($v[\mathbb{T}]$) and a subformula (ϕ_1). An existentially quantified formula is satisfied if and only if for some domain element d in the type $\mathbb{T}$, $\phi_1\{v \mapsto d\}$ is satisfied. We also say that an existentially quantified formula *binds* the variable v. We use

$$\exists v_1 \dots v_n : \phi_1.$$

or

$$\exists \bar{v} : \phi_1.$$

as a shorthand for

$$\exists v_1 : (\exists v_2 : (\dots (\exists v_n : \phi_1) \dots)).$$

For existentially quantified formulas, we also allow expressions of the following form, with n a natural number.

$$\exists_{=n} v : \phi_1.$$

These expressions are satisfied if and only if, for *exactly* n different domain elements d in the type $\mathbb{T}$, $\phi_1\{v \mapsto d\}$ is satisfied. We also provide similar expressions of the form $\exists_{>n}$, $\exists_{\leq n}$, and $\exists_{\geq n}$. If $n > 0$, we also provide $\exists_{<n}$.

The variables v in the quantified formulas are called the "quantified variables", or "scoped variables". We also say that these formulas "quantify" over v. A variable is *bound* by its innermost occurrence as a quantified variable. A formula that has no free variables is called a *sentence*.

In this text we also allow the usage of $\phi_1 \Rightarrow \phi_2$ as a shorthand for $\neg\phi_1 \vee \phi_2$ and of $\phi_1 \Leftrightarrow \phi_2$ as a shorthand for $(\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$.

These combinators for formulas have the following binding precedence (strongest-to-least binding):

(1) brackets (between "(" and ")")

(2) negation ($\neg$)

(3) conjunction ($\wedge$)

(4) disjunction ($\vee$)

(5) any quantifier ($\forall$ or $\exists$).

Example 2.2.12. A formula of the form

$$\exists x : \neg\forall y : P(x, y) \vee Q(y, x).$$

expresses the following, with round brackets around formulas indicating priority binding

$$\exists x : (\neg\forall y : (P(x, y) \vee Q(y, x))).$$

Definition

A definition (Δ) is used to *define* a (set of) predicate(s). As shown in Listing 2.9, a definition occurs between curly brackets "{" and "}". A definition contains a set of rules of the form

$$\forall \bar{v} : P(\bar{v}) \leftarrow \phi[\bar{v}].$$

We call the $P(\bar{v})$ the *head* of the rule and $\phi[\bar{v}]$ the *body* of the rule. All predicate symbols that occur in the head of any of the rules in a definition are called the *defined* predicates of that definition. Symbols that only occur in the body of the rules are called the *open* symbols of that definition.

As a terminology shortcut we sometimes speak of the rules present in a theory. With this, we mean the set of rules that is present in any of the definitions in the theory.

Because the semantics for defining functions is still ongoing work, we limit ourselves to defined *predicates* in this text. The IDP3 system supports defining functions, but uses the semantics of the "corresponding" predicate. In recent years there were some proposals [7] to define semantics for functions in the head of a rule.

Set Expressions

A set expression is an expression of the form $\{\bar{v} : \phi[\bar{v}] : t[\bar{v}]\}$ with ϕ any complex formula. The meaning of a set expression is given in Section 2.3.1.

2.2.4 Concluding Example: Sudoku

In this section we wrap up our explanation of the FO(·) language by presenting a full example of an FO(·) encoding of a sudoku problem. The encoding (shown in Listing 2.10) could be simplified in different ways, but a verbose one is chosen for increased clarity.

The vocabulary Σ contains three types that are also natural numbers: indices (`Index`), values (`Value`), and blocks (`Block`). Next is a predicate that represents the relation of indices to blocks (`InBlock`). Finally, there is a function (`ValueOf`) that maps coordinates on the grid to values. It is considered good practice to specify the intuitive meaning of symbols in the vocabulary where they are declared, as is done in Listing 2.10.

The interpretation I provides a partially filled-in sudoku. A graphical representation of this is shown in Figure 2.1.

The theory $\mathcal{T}$ contains a definition (of the `InBlock` predicate) and three formulas, tagged with numbers 1 through 3. The definition states that for a square in row r and column c, its block number is defined by the expression

$$((r-1) - (r-1)\%3) + ((c-1) - (c-1)\%3)/3 + 1.$$

```
vocabulary Σ {
  type Index isa nat    // x- or y-index on the sudoku grid
  type Value isa nat    // values for inside a sudoku square
  type Block isa nat    // the 3x3 big squares
  InBlock(Index,Index,Block) // which squares are what blocks
  ValueOf(Index,Index):Value // mapping of squares to values
}

structure I : Σ {
  Index = {1..9}
  Value = {1..9}
  Block = {1..9}
  ValueOf<ct> = { 1,1->4; 3,3->6; 7,8->2; 9,3->1}
  ValueOf<cf> = { 2,2->4; 4,4->5}
}

theory T : Σ {
  ∀ r[Index] v[Value]: ∃=1 c[Index]: ValueOf(r,c) = v. // (1)
  ∀ c[Index] v[Value]: ∃=1 r[Index]: ValueOf(r,c) = v. // (2)
  ∀ b v : ∃=1 r c:InBlock(r,c,b) ∧ ValueOf(r,c) = v. // (3)
  {
    ∀ c r b : InBlock(r,c,b) ←
                b = ((r-1) - ((r-1) % 3))
                  + ((c-1) - ((c-1) % 3))/3
                  + 1.
  }
}
```

Listing 2.10: An example FO(·) encoding of a Sudoku problem instance

Formula `(1)` states that for every row r and value v, there has to be exactly one column on row r with value v. Formula `(2)` states that for every column c and value v, there has to be exactly one row on column c with value v. Formula `(3)` states that every block b and value v, there has to be exactly one square in that block with value v.

We use the notation $v[\mathbb{T}]$ to indicate that the variable v has type $\mathbb{T}$. In this manner, the types associated with the variables in formulas `(1)` and `(2)` are stated explicitly. However, type information can also be left out if a *type derivation system* is able to derive the correct typings for variables. As an example of this, we show formula `(3)` without explicit typings.

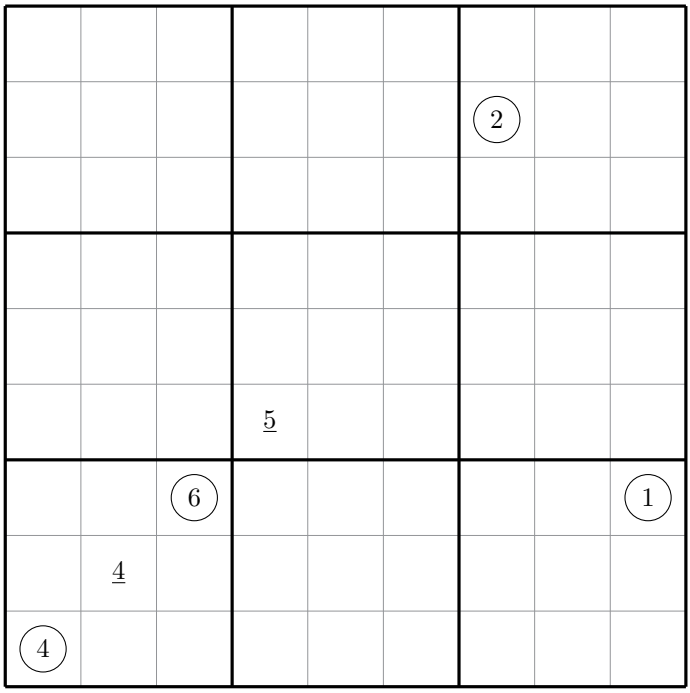


Figure 2.1: A graphical representation of the input structure shown in Listing 2.10. Underlined numbers indicate a value that must not be placed in the square. Circled numbers indicate a value that must be placed in the square.

2.3 Semantics for $\mathrm{FO}(\cdot)$

Given the constructs of a theory and an interpretation, we now associate meaning to the theory by defining the satisfaction relation. Given a theory $\mathcal{T}$ and a structure I, we use

$$I \models \mathcal{T}$$

to denote that all desired properties and restrictions imposed by $\mathcal{T}$ are satisfied in I. Let $\mathcal{T}^{\phi}$ be the set of formulas in $\mathcal{T}$ and $\mathcal{T}^{\Delta}$ the set of definitions in $\mathcal{T}$. A theory $\mathcal{T}$ is satisfied in an interpretation I, if all formulas in $\mathcal{T}$ and all definitions in $\mathcal{T}$ are satisfied in I. I.e., $I \models \mathcal{T}$ if and only if $\forall \phi \in \mathcal{T}^{\phi} : I \models \phi$ and $\forall \Delta \in \mathcal{T}^{\Delta} : I \models \Delta$. Section 2.3.1 specifies how to evaluate a term in a structure, Section 2.3.2 gives the satisfaction relation for formulas, and Section 2.3.3 specifies the satisfaction relation for definitions.

2.3.1 Evaluating a Term in a Structure

A built-in value val always evaluates to val: $val^I = val$. A variable v is substituted with the term it is bound to when evaluating: $v\{v \mapsto t, v' \mapsto t', \dots\}^I = t^I$. A function term $F(t_1, \dots, t_n)^I$ evaluates to t_{out} if and only if $F^I(t_1^I, \dots, t_n^I, t_{out}) = \texttt{true}$.

Let $\texttt{agg}_t$ be an aggregate function ($\texttt{agg}_f$) applied to a setexpression (E). An aggregate term $\texttt{agg}_t$ is evaluated by evaluating its associated set expression and applying the aggregate function to it. A set expression is an expression of the form $\{\bar{v} : \phi[\bar{v}] : t[\bar{v}]\}$ with ϕ any complex formula. This expression is evaluated w.r.t. I to a multiset of terms in the following manner: **(1)** start with an empty multiset S, **(2)** find bindings for $\bar{v}$ such that $I \models \phi\{v_1 \mapsto t_1, \dots v_n \mapsto t_n\}$, **(3)** for all these bindings, evaluate the term $t\{v_1 \mapsto t_1, \dots v_n \mapsto t_n\}$ and insert its evaluation into S, and **(4)** return S. An aggregate function is one of $\{\texttt{min}, \texttt{max}, \texttt{card}, \texttt{sum}, \texttt{prod}\}$ that maps this resulting multiset to its minimal element, maximal element, number of elements, sum of its elements, or product of its elements.

2.3.2 Semantics of Formulas

The satisfaction relation of a formula is defined inductively according to Tarski's truth definitions [75, 76].

- $I \models P(t)$ if and only if $P^I(t^I) = \texttt{true}$.
- $I \models \phi_1 \wedge \phi_2$ if and only if $I \models \phi_1$ and $I \models \phi_2$.
- $I \models \phi_1 \vee \phi_2$ if and only if $I \models \phi_1$ or $I \models \phi_2$.
- $I \models \neg\phi$ if and only if $I \models \phi$ does not hold.
- $I \models \forall v[\mathbb{T}] : \phi$ if and only if, for **every** domain element d in type $\mathbb{T}$, $I \models \phi\{v \mapsto d\}$.
- $I \models \exists v[\mathbb{T}] : \phi$ if and only if, for **some** domain element d in type $\mathbb{T}$, $I \models \phi\{v \mapsto d\}$.

2.3.3 Semantics of Definitions

The satisfaction relation of a definition is less straightforward. The Logic Programming (LP) community has considered different semantics for logic

$$\left\{ \begin{array}{l} p \leftarrow \neg q. \\ q \leftarrow \neg p. \end{array} \right\}$$

Figure 2.2: An example of a "loop over negation". In this definition, p depends negatively on q, and q depends negatively on p, resulting in two stable models: $\{p\}$ and $\{q\}$.

programs throughout its history. It has been established that Logic programs can be seen as inductive definitions [28], so semantics for logic programming can be used as semantics for inductive definitions. Perhaps the most known of these semantics are the Stable Semantics [46] and the Well-Founded Semantics (WFS) [81]. In Answer Set Programming (ASP), the Stable Semantics is used as a basis for the semantics of non-determinism [46, 62]. In short, "loops over negation" such as the one shown in Figure 2.2 are required to provide non-determinism when modeling. These loops over negation then lead to multiple possible models, each one making a choice over which element in the loop is made true. Although syntactic sugar is provided through the use of *choice rules* [5], the dependency on Stable Semantics remains.

In IDP3, the Well-Founded Semantics is used. The major motivation for this choice is the fact that it has been argued many times over [26, 31, 33] that the Well-Founded Semantics coincides with the natural semantics associated with inductive definitions present in mathematical texts. For declarative solving based on a Knowledge Representation (KR) perspective, this argument is important. In addition to this, FO(·) introduces non-determinism by allowing values to be set as `unknown` in interpretations, omitting the need for Stable Semantics. See Section 2.2.2 for examples of how something "unknown" is presented. For an in-depth comparison between ASP and FO(·) we refer to more detailed work [30, 35]. In addition to this, there is also work [51] that shows the link between three-valued logic and two-valued logic in ASP, further bridging the gap.

For a more in-depth description of the Well-Founded Semantics we refer to its first introduction by Van Gelder et al. [81]. Additionally, there is more recent work that has used the concept of *justifications* to specify several semantics, including the Well-Founded Semantics [27, 29]. Our chapter on relevance (see Chapter 6) also includes an explanation of justifications and how they are used to specify the WFS on definitions.

2.4 Conclusion

This chapter contains a description of the mathematical concepts used in this thesis in Section 2.1. A description of the FO(·) language was given: Section 2.2 introduces the syntax and Section 2.3 discusses the semantics.

Chapter 3

Preliminaries: KBS and IDP

This chapter and the previous one serve as a summary of all the background knowledge that is assumed in the remainder of this text. In Section 3.1 we present the Knowledge Base System (KBS) paradigm as an alternative to more classical software development techniques. Section 3.2 contains a detailed description of the IDP3 system that provides an implementation of the KBS paradigm based on the language FO(·).

3.1 KBS paradigm: Inferences as Tools for Solving Problems

Given the syntax and semantics of the FO(·) language described in the previous chapter, we show how these can be used as an alternative approach to software development. This alternative approach is labeled the Knowledge Base System (KBS) paradigm and it aims to provide an alternative software development methodology. This section contains a short description of the KBS paradigm. A more extended presentation is given by Denecker and Vennekens [32].

The KBS paradigm philosophy exploits the richness and compactness of the FO(·) language to heavily focus on a natural representation of knowledge, as well as the ability to reuse that knowledge in different ways for different tasks. As a first part, the FO(·) language is used to represent the *core* aspects of a problem domain, the (possibly quite complex) knowledge that is associated with that problem. These FO(·) specifications (i.e., theories, structures, and vocabularies) can then be used to solve tasks by applying *inferences* to them.

In Section 3.1.1 we give an overview of the inferences and in Section 3.1.2 we show the benefit of this approach.

3.1.1 Inferences

An inference is a base task that, depending on the FO($\cdot$) specification and its semantics, gives some useful output related to the problem at hand. In this section we give a definition of the following inferences:

(1) model expansion,

(2) model checking,

(3) model revision, and

(4) unsat core generation.

In the definitions below, we assume **(1)** that Σ is a vocabulary, **(2)** $\mathcal{T}$ is a theory over Σ, and **(3)** I is an interpretation over Σ.

Definition 3.1.1 (The model expansion inference)**.** The *model expansion inference* (denoted as `mx`) takes as input a theory and a structure, and produces a set of two-valued structures that are refinements of I that satisfy $\mathcal{T}$. More formally:

$$I' \in \texttt{mx}(\mathcal{T}, I) \iff I \leq_p I' \wedge I' \models \mathcal{T}$$

The model expansion inference is used to find a completion (I') for a given partial assignment (I) that satisfies the restrictions in $\mathcal{T}$.

Example 3.1.2. Given the sudoku solution specification for a sudoku problem in Listing 2.10, calling $\texttt{mx}(\mathcal{T}, I)$ results in a set containing two-valued interpretations that have all squares of the sudoku filled-in. A graphical representation of such a possible solution is shown in Figure 3.1.

Definition 3.1.3 (The model checking inference)**.** The *model checking inference* (denoted as `mcheck`) takes as input a theory and a structure and returns a Boolean value `true` or `false` that indicates whether or not $I \models \mathcal{T}$. More formally:

$$\texttt{mcheck}(\mathcal{T}, I) = \texttt{true} \iff I \models \mathcal{T}$$

The model checking inference is used to check whether a two-valued assignment satisfies all restrictions in some theory $\mathcal{T}$.

2	3	1	8	6	4	7	5	9
5	4	8	9	7	1	2	6	3
9	6	7	5	2	3	4	1	8
3	2	5	1	9	6	8	7	4
6	7	4	3	8	2	1	9	5
1	8	9	4	5	7	6	3	2
8	5	6	7	4	9	3	2	1
7	9	3	2	1	8	5	4	6
4	1	2	6	3	5	9	8	7

Figure 3.1: A graphical representation of I_s.

Example 3.1.4. Given the sudoku solution I_s shown in Figure 3.1, calling `mcheck`$(\mathcal{T}, I_s)$ returns `true`.

The above two inferences will be focused on in the remainder of this text. We give a short description of some additional inferences, to provide some more examples of inferences available to the KBS paradigm.

Definition 3.1.5 (The query inference)**.** The *query inference* (denoted as `query`) takes as input a two-valued structure I, and a query expression of the form $\{v_1, \ldots, v_n : \phi\}$. It returns a set of all lists $(d_1, \ldots, d_n)$ such that

$$\phi\{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}^I = \texttt{true}.$$

Definition 3.1.6 (The model revision inference)**.** The *model revision inference* (denoted as `mrev`) takes as input a theory $\mathcal{T}$ and a two-valued structure I. It returns a two-valued structure that satisfies $\mathcal{T}$ that is "close" to I in some proximity metric.

Definition 3.1.7 (The unsat core generation inference)**.** The *unsat core generation inference* (denoted as `unsatcore`) takes as input a theory $\mathcal{T}$ and a

structure I for which $\mathcal{T}$ is not satisfied. The inference returns a sub-theory $\mathcal{T}'$ of $\mathcal{T}$ that is not satisfied in I that is as small as possible.

The unsat core generation inference is used to investigate *why* a certain structure does not satisfy a given $\mathcal{T}$. This is useful during *debugging* of FO$(\cdot)$ specifications; it could be that a formula was written down incorrectly, or that the input structure was not properly formulated. Using unsat core generation helps the end-user to find encoding errors in FO$(\cdot)$ specifications.

3.1.2 Advantages of the KBS Paradigm

The KBS paradigm is a *declarative* paradigm. This means that instead of specifying *how* something must be solved, one specifies *what* the problem is and lets the system figure out how to solve it [4, 43]. This results in software solutions to problems that sometimes reduce the code base by 90% [14]. The KBS paradigm most clearly presents its advantages in applications where either **(1)** the knowledge related to the problem changes rapidly (i.e., tax form restrictions that may change every year), **(2)** the knowledge related to the problem is very complex (i.e., employee scheduling problems for a large corporation), or **(3)** it is very difficult to define a procedural approach to solving the problem (i.e., planning problems)

Because the KBS contains an explicit representation of the restrictions and information related to the problem, this representation is smaller and easier to understand. In addition to this, an FO$(\cdot)$ specification can be reused in several ways for multiple inferences. Consider a theory that represents the tax form restrictions. This theory can be used to check whether a given tax form (which is the structure in this case) respects the restrictions. The same theory can then be used to complete a partially filled-in tax form, ensuring that it is completed in a legally correct manner.

3.2 The IDP System

The IDP3 system is an implementation that supports programming according to the KBS paradigm. This section gives a high-level explanation to the system and the more important techniques that are present in it. References to detailed papers are provided where appropriate. The explanation of IDP3 (usually just denoted with IDP) in this section is a representation of the system at the end of the year 2013, excluding amendements developed in this thesis project. In

the following chapters several additions and improvements to the IDP3 system are presented.

The IDP system is a ground-and-solve system, which means it can be divided up into two parts: the *grounder* and the *solver*. The solver of the IDP system is called MiniSAT(ID) [65] and is an extension of the SAT solver MiniSAT [36]. The grounder is responsible for taking an FO(·) theory and structure as input and transforming it into a more low-level representation that does not contain variables. The solver then takes this low-level representation as an input and calculates the required output.

Example 3.2.1. The required output of the solver changes depending on the inference. For the model expansion inference, a two-valued interpretation must be provided by the solver. For the model checking inference, a simple `true` or `false` answer is required.

In the following sections we give a more in-depth representation of the grounder and the solver respectively. A more thorough introduction and manual on how to use the system is provided by De Cat et al. [19]

3.2.1 The Grounder of IDP

This section provides an explanation of the grounder of IDP3 and a high-level, intuitive presentation of its most important incorporated techniques. This section is largely based on the first part of earlier published work of the author of this text [53]. The discussion in this section is focused on the grounder of IDP3, but several other grounding approaches exist in comparable state-of-the-art declarative systems.

- Lparse [74] is a grounder for logic programs with the stable model semantics. Lparse also employs techniques such as *local grounding* to limit the memory overhead and the division of predicates into *domain predicates* (value generators, comparable to the types in IDP3) and *non-domain predicates* (IDP3 predicates).
- Gringo [45] and DLV [60] are grounders for ASP programs. They ground using semi-naive evaluation and employ techniques such as the creation of dependency graphs on the predicate level to improve performance.

The grounder of IDP transforms the input FO(·) specification to a theory in Extended Conjunctive Normal Form (ECNF), which is the input of the solver. The input FO(·) specification consists of an input theory $\mathcal{T}$ and an

input structure I. This transformation is also called the process of "grounding". Thus, IDP's grounder transforms the formulas and definitions in $\mathcal{T}$ to clauses in an ECNF theory with respect to the given structure I.

This conversion in turn has two core elements: *instantiating* quantifiers, and *flattening* nested propositional formulas.

Example 3.2.2. Given a sentence

$$\phi = \exists v[\mathbb{T}] : P(v) \wedge Q(v)$$

with $\mathbb{T}^I = \{d_1, d_2\}$, then ϕ's instantiation is

$$inst(\phi) = (P(d_1) \wedge Q(d_1)) \vee (P(d_2) \wedge Q(d_2)).$$

To turn $inst(\phi)$ into a Conjunctive Normal Form (CNF), we would need to flatten it. This can be efficiently done by introducing helper symbols called *Tseitin literals* [78] that are in essence a reification of a nested subformula:

$$\begin{aligned} flat(inst(\phi)) =& (L(d_1) \vee L(d_2)) \\ & \wedge (L(d_1) \Leftrightarrow P(d_1) \wedge Q(d_1)) \\ & \wedge (L(d_2) \Leftrightarrow P(d_2) \wedge Q(d_2)). \end{aligned}$$

Now, turning the $\Leftrightarrow$-formulas into clauses results in a grounded version of ϕ:

$$\begin{aligned} flat(inst(\phi))' =& (L(d_1) \vee L(d_2)) \\ & \wedge (\neg L(d_1) \vee P(d_1)) \\ & \wedge (\neg L(d_1) \vee Q(d_1)) \\ & \wedge (L(d_1) \vee \neg P(d_1) \vee \neg Q(d_1)) \\ & \wedge (\neg L(d_2) \vee P(d_2)) \\ & \wedge (\neg L(d_2) \vee Q(d_2)) \\ & \wedge (L(d_2) \vee \neg P(d_2) \vee \neg Q(d_2)). \end{aligned}$$

This grounding process assumes a set of normalized first-order logic sentences as input. Normalizing a sentence (i.e., reducing a sentence to a normalized format) consists of unnesting function symbols (see Section 5.2.1 for a detailed explanation), transforming function symbols to their graph predicate, pushing quantifiers inwards, and pushing negations through quantifiers [23].

Finally, IDP3 and its language FO($\cdot$) support more constraints than only first-order sentences: inductive definitions [31], sum constraints, product constraints and cardinality constraints [21] are the most noteworthy. Recent work [20, 64] on MiniSAT(ID) added support for grounded versions of these features, and the ECNF language specification has since been extended. For the sake of clarity we also ignore these extra features in the remainder of this exposition of the grounder and reduce our discussion to the grounding of first-order sentences.

So to summarize, we consider a grounding task that consists of transforming a structure and a theory containing only first-order sentences, to a conjunction of clauses called the ground theory or *grounding*. The first-order sentences are normalized, but not yet instantiated or flattened.

To control the complexity of this task, IDP3 uses the following three grounding optimization techniques: Reduced Grounding (RED) [21], Lifted Unit Propagation (LUP) [87], and Grounding With Bounds (GWB) [88]. Together, these techniques provide three advantages. They refine the input structure I, reduce the size of the resulting ground theory, and reduce the time needed to ground.

During this exposition of the grounder, the concept of the *size* of a ground theory is needed. We define the size of a ground theory as the number of ground atoms in the ground theory. Since we assume our ground theory to be a conjunction of clauses, the size of the theory is the sum of the size of each clause. For instance, the size of $flat(inst(\phi))'$ in Example 3.2.2 is 16.

In this section a high-level explanation is given for the three mentioned grounding techniques RED, LUP, and GWB. The aim of this section is to give the reader an intuition as to what these techniques achieve and the impact they have on each other. Each of these techniques has been described in detail in other publications and references are provided.

Reduced Grounding

The intuition for the Reduced Grounding (RED) [21] technique is that (sub)formulas of which we know the truth value beforehand do not have to be grounded and can be substituted with their truth value in the given interpretation. Whenever the top-down grounder enters a leaf containing an atom whose truth value is known, that value is filled in. We call this an *evaluation* of the atom; known domain atoms are substituted with their truth value. Additionally, when a formula has a subformula whose truth value is known, this truth value is propagated for this formula. E.g., if one of the disjuncts in a disjunction is true, that entire disjunction is true as well. This is

$$\begin{array}{rcl|rcl} \neg\texttt{true} & \rightarrowtail & \texttt{false} & \neg\texttt{false} & \rightarrowtail & \texttt{true} \\ \phi \vee \texttt{true} & \rightarrowtail & \texttt{true} & \phi \wedge \texttt{false} & \rightarrowtail & \texttt{false} \\ \forall v[\mathbb{T}] : \texttt{true} & \rightarrowtail & \texttt{true} & \exists v[\mathbb{T}] : \texttt{true} & \rightarrowtail & \texttt{true} \\ \forall v[\mathbb{T}] : \texttt{false} & \rightarrowtail & \texttt{false} & \exists v[\mathbb{T}] : \texttt{false} & \rightarrowtail & \texttt{false} \end{array}$$

Figure 3.2: Some simplification rules.

called *simplification* [21]. Some of the rules used in simplifications are shown in Figure 3.2. Note that RED can only start working after a leaf of the top-down grounding process has been encountered.

Example 3.2.3. Consider the formula

$$\exists v[\mathbb{T}] : P(v) \vee \forall v'[\mathbb{T}] : \phi(v, v') \tag{3.1}$$

with $\mathbb{T}^I = \{1, 2, 3\}$, and $P^I(1) = \texttt{true}$. Since there are two quantified variables, the naive grounding has a size of order n^2 with n the number of elements in $\mathbb{T}^I$. Using the above technique we show how the grounding can be simplified to the atom `true`.

Formula (3.1) is grounded by iterating over the domain elements in $\mathbb{T}^I$, instantiating it, and grounding the subformula. In order to improve memory usage, we use a depth-first approach, which means the first instantiation for v has to be ground before more instantiations for v are considered for grounding. Assume we start by instantiating v with 1, which means the grounder will continue by grounding the first (instantiated) disjunct in (3.2).

Now the grounder encounters (3.3) and grounds it by grounding each of the disjuncts, the order of which is not specified. Assume we start by grounding the first disjunct, the atom $P(1)$. We know the truth value of this atom, because $P^I(1) = \texttt{true}$, leading to the formula (3.4). The simplification rule $\phi \vee \texttt{true} \rightarrowtail \texttt{true}$ is applicable and the grounder simplifies the entire disjunction to `true`. This is returned as the result for the first disjunct in formula (3.2), leading to formula (3.5). Following the same simplification rule, this entire formula can again be simplified to `true`, shown in (3.6). Note that we assumed we were "lucky" enough to first instantiate v with 1 in formula (3.1) and to select the first disjunct when grounding formula (3.3). There is however no guarantee that this happens and an alternative run of the grounder could end up grounding $\forall v'[\mathbb{T}] : \phi(v, v')$ first, before finding out simplifications that can be made. This shows the unpredictable nature of the benefits of this approach.

A note on implementation Reduced grounding is considered the most straightforward technique for grounding. Given that the implementation of

$$(P(1) \vee \forall v'[\mathbb{T}] : \phi(1, v')) \vee \exists v[\mathbb{T}\backslash\{1\}] : P(v) \vee \forall v'[\mathbb{T}] : \phi(v, v') \tag{3.2}$$

$$P(1) \vee \forall v'[\mathbb{T}] : \phi(1, v') \tag{3.3}$$

$$\texttt{true} \vee \forall v'[\mathbb{T}] : \phi(1, v') \tag{3.4}$$

$$\texttt{true} \vee \exists v[\mathbb{T}\backslash\{1\}] : P(v) \vee \forall v'[\mathbb{T}] : \phi(v, v') \tag{3.5}$$

$$\texttt{true} \tag{3.6}$$

Figure 3.3: Different intermediate formulas when grounding (3.1) with RED enabled.

this technique is essentially the application of a variety of substitution rules, implementing it is considered to be no great challenge. Compared to the other two techniques we will discuss, it is fair to say that this one is the easiest to implement.

Lifted Unit Propagation

Lifted unit propagation (LUP) [87] is a technique that aims to refine the input structure I without losing any models. More formally, it aims to produce a structure I' such that **(1)** $I \leq_p I'$ and **(2)** $I' \models \mathcal{T}$.

This is done by creating a symbolic representation of the theory $\mathcal{T}$ containing the truth dependencies between formulas in $\mathcal{T}$. More specifically, the symbolic representation expresses for each formula when it can be derived to be certainly true (`CT`) or certainly false (`CF`), depending on the `CT` or `CF` information about its sub- or superformulas. An example of this is that if a disjunction is known to be `CF`, each of disjuncts has to be `CF` as well. Using the symbolic representation, we query for all domain atoms which instances can be derived to be `CT` or `CF`. This information is then used to create the refined structure I'.

Example 3.2.4. Consider the theory containing formula ϕ shown in (3.7) with I specifying $\mathbb{T} = \{1, 2, 3\}$ and $P^I(1) = \texttt{true}$.

$$\forall v[\mathbb{T}] : P(v) \Rightarrow Q(v) \tag{3.7}$$

For this theory we give part of the symbolic representation as a definition in Figure 3.4 and use $\phi'(v)$ to denote the subformula of ϕ shown in (3.8).

$$P(v) \Rightarrow Q(v) \tag{3.8}$$

We show that by propagating truth values, one can derive that $Q(1)$ is true. Since ϕ is a top-level formula it has to be true in order to satisfy the theory.

$$\left\{\begin{array}{rcl} \phi_{\mathtt{CT}} & \leftarrow & \mathtt{true} \\ \phi'_{\mathtt{CT}}(v) & \leftarrow & \phi_{\mathtt{CT}} \\ Q_{\mathtt{CT}}(v) & \leftarrow & \phi'_{\mathtt{CT}}(v) \wedge P_{\mathtt{CT}}(v) \\ \phi'_{\mathtt{CT}}(v) & \leftarrow & P_{\mathtt{CF}}(v) \\ \phi'_{\mathtt{CT}}(v) & \leftarrow & Q_{\mathtt{CT}}(v) \\ & \cdots & \end{array}\right\}$$

Figure 3.4: Example of a symbolic representation for formula (3.7).

This is expressed in the first rule. The second rule states that if the universally quantified formula $\forall v : \phi'$ is true, ϕ' each instance of the subformula ϕ' has to be true as well. The third rule expresses that when it is known that $\phi'(v)$ is true and $P(v)$ is true, formula (3.8) forces that $Q(v)$ is true as well. Because $P(1)$ is known to be true in I, and all $\phi'(v)$ are true because ϕ is a top-level formula, $Q(1)$ will be returned when the symbolic representation is queried for values for which it is known that $Q(v)$ is true. With I' the newly created refined structure, $Q^{I'}(1) = \mathtt{true}$, which means that $I \leq_p I'$. We lose no models because $Q^M(1) = \mathtt{true}$ holds for any model M of $\mathcal{T}$ that is a refinement of I.

When applying this technique without any of the two other techniques, the resulting grounding will not change, since the information present in I' is not exploited in the grounding process. The benefit of this technique lies in the additional information provided in I' that can be exploited by other grounding techniques, such as RED, that use information present in the structure.

A note on implementation This technique works for any symbolic representation of the theory. We consider the implementation suggested by Wittocx et al. [86], which is based on Binary Decision Diagrams (BDDs). However, as is mentioned by Wittocx et al., a "complete" symbolic representation of the theory can be very complex and the associated calculation computationally expensive. Because of this, an approximative implementation is considered that places limitations on the complexity of the symbolic representation of the theory. More specifically, the BDDs that are used are limited in the length of their branches and when a BDD becomes too big, it is pruned (i.e., branches are cut off and removed), leading to an implementation that is approximative. This approximativeness means that not **all**, but instead **some** structure refinements are made. Correctness is not jeopardized.

input : A theory T and a partial structure I
output : A quantifier-free theory equisatisfiable with T
Function *groundTheory(T,I)*:
  $T' \leftarrow$ transform(T)
  $S^t \leftarrow$ tseitsinize(T')
  $G \leftarrow \emptyset$
  $I' \leftarrow$ LUP(I,S^t)
  for *formula* ϕ *in* S^t **do**
    $G \leftarrow G \cup$ groundForm(ϕ, I')
  return toCNF(G)

Algorithm 1: High level overview of the workflow.

Grounding With Bounds

Ground With Bounds (GWB) [88] is a technique that detects when formulas are already `true` or `false` *before* grounding them. Recall that in Example 3.2.3 we mentioned that the benefits of RED depend greatly on the order in which formulas are grounded, since it needs to evaluate a domain term in a leaf of the grounding tree before being able to propagate (simplify) this information upwards. In contrast to this, GWB tries to detect propagations that RED can do *before* arriving at these leaf atoms. This allows GWB to offer the benefits of RED without depending on the order of grounding.

GWB uses a symbolic representation of the theory that is closely related to the one used in LUP. The only difference is that it does not contain rules deriving that top-level formulas are true by default, such as the first rule in Figure 3.4. Using this symbolic representation, we can query each formula for instances that are known to be `CT` or `CF`. Instances of a formula that are known to eventually simplify to `true` (`false`) are called the `CT` (`CF`) *bound* for this formula, hence the name *grounding with bounds*. Formally, a bound for a formula is a set of assignments to the free variables in that formula that makes its interpretation `true` (for a `CT` bound) or `false` (for a `CF` bound). If the set of assignments in the bound is *larger*, this bound is more precise and we denote it as a *tighter* bound on the formula. These bounds are used to limit which subformulas, as well as which instances for quantified variables, are considered for grounding. When GWB is used in the workflow, it reduces the amount of subformulas that need to be grounded. For a conjunction this is the number of conjuncts, for a quantifier this is the instances of the quantified variables for which the subformula is handled.

Example 3.2.5. We use the sentence shown in Example 3.2.4. Consider the theory containing formula ϕ shown in (3.7) and assume I contains the

information $\mathbb{T} = \{1, 2, 3\}$, $P^I(1) = \texttt{true}$, and $Q^I(1) = \texttt{true}$. When instantiating the universally quantified variable v in formula ϕ, we are only interested in values for which the subformula ϕ' is not known to be true. This is because the grounding for these instantiations would end up being simplified to `true` anyway, meaning that these can be dropped from the large conjunction that the grounding ϕ would become. Therefor, we are only interested in values of v for which it is not yet known that $\phi'(v)$ is true. This corresponds with querying our symbolic representation with $\{v : \neg\phi'_{\texttt{CT}}(v)\}$. We refer again to the symbolic representation as a definition in Figure 3.4 and observe that rule five expresses that the implication is trivially satisfied if its consequent is true. Since $Q(1)$ is known to be true, $\phi'(1)$ is derived to be true, eliminating it from the answer set of the above query. The answer set is then $\{2, 3\}$, meaning that we ground $\forall v[\mathbb{T}] : \phi'(v)$ into $\phi'_g(2) \wedge \phi'_g(3)$ with ϕ'_g indicating that we recursively ground the remaining conjuncts.

A note on implementation Similar to the implementation of LUP above, we use a symbolic representation of the theory that is based on BDDs and is approximative.

Overview of the workflow

Algorithms 1 and 2 illustrate the high-level structure of our top-down, depth-first grounding algorithm. The three discussed techniques are integrated into the grounding workflow. Lifted Unit Propagation is called before the grounding of the individual formulas (Algorithm 1, line 5), resulting in I' with $I \leq_p I'$. Ground With Bounds is called at the start of every recursive call (Algorithm 2, line 2), eliminating parts of the formula that are unnecessary to ground by reducing the domain of quantified variables). The Reduced Grounding is the combination of the simplify at the end of every recursive call (line 23) and the evaluation of atoms (line 6) in Algorithm 2; this helps reducing the grounding after it is made. More interested readers can find the source code of the grounder discussed here as part of the IDP3 system at `http://dtai.cs.kuleuven.be/krr/software/idp`

input : Formula ϕ, structure I
output : A quantifier-free and possibly simpler version of ϕ

Function *groundForm(ϕ,I)*:
  $\phi' \leftarrow \mathrm{GWB}(\phi)$
  switch ϕ' **do**
    case *atom $P(\bar{v})$* **do**
      if *I contains $P(\bar{v})$* **then**
        return evaluate($\mathrm{P}(\bar{v})$,I)
      else
        return $\mathrm{P}(\bar{v})$
    case $\bigvee_i \psi_i$ **do**
      $\mathrm{d} \leftarrow \emptyset$
      for ψ_i *in* ϕ' **do**
        $d \leftarrow d \cup \mathrm{groundForm}(\psi_i, I)$
      out $\leftarrow \mathrm{disjunction}(d)$
    case $\bigwedge_i \psi_i$ **do**
      $\cdots$ (analog to the disjunctive case)
    case $\exists v[\mathbb{T}] : \psi$ **do**
      $\mathrm{d} \leftarrow \emptyset$
      for `value` $\in \mathbb{T}$ **do**
        $d \leftarrow d \cup \mathrm{groundForm}(\psi\{v \mapsto \texttt{value}\}, I)$
      out $\leftarrow \mathrm{disjunction}(d)$
    case $\forall v[\mathbb{T}] : \psi$ **do**
      $\cdots$ (analog to the existential case)
  return simplify(*out*)

Algorithm 2: The grounding of a formula w.r.t. a structure.

3.2.2 The Solver of IDP

This section provides an explanation of the solver of IDP and a high-level, intuitive presentation of its most important incorporated techniques.

The solver of IDP (MiniSAT(ID) [65]) supports input in the Clausal Normal Form (CNF), Extended Conjunctive Normal Form (ECNF), Quantified Boolean Form (QBF, CNF's higher-order relative), ground ASP (in the LParse-Smodels intermediate format [74]) and FlatZinc [68]. The solver can be seen as a standalone component that processes the above input. If used in conjunction with the grounder of IDP, the solver uses the ECNF format. The task of the solver is to either **(1)** check whether the given specification is satisfiable or **(2)** return a two-valued assignment that satisfies the given specification.

In the remainder of this subsection we enumerate the important techniques that are present in SAT solvers. More information in regard to the basic SAT techniques is provided by Biere et al. [13]. For our explanation, we start from the SAT context and extend it with extra features that have been implemented in MiniSAT(ID). The performance of MiniSAT(ID) has been investigated by Amadini et al. [2, 3], where it turned out to be the single-best solver in their MiniZinc portfolio.

Basic SAT (i.e., Propositional Logic)

Formal specifications written in First-Order Logic (i.e., a restricted form of FO($\cdot$) without types, functions, definitions ...) can be reduced to CNF. SAT solvers take CNF as input. Figure 3.5 shows an example of input in the Conjunctive Normal Form (CNF). A CNF contains a set of predicate applications without variables, also called the *propositional variables*. The propositional variables that can occur in the specification (in this example $\{a, b, c, d\}$) are also called the *variables* for short. A *literal* is the occurrence of a possibly negated variable (in this example $\{a, \neg a, b, \neg b, c, \neg c, d, \neg d\}$). A negated variable is also called a *negative* literal. Otherwise, it is called a *positive* literal. A *clause* is a disjunction of literals. A CNF specification consists of a conjunction of clauses. An *assignment* is a partial mapping of variables to truth values $\{\texttt{true}, \texttt{false}\}$ that considers the positive literal (resp. negative literal) constructed from this variable to be true. If the assignment in the solver does not map a variable, that variable is called *unassigned*.

In order for the CNF to be satisfied, at least one literal in each clause must be true in the assignment.

$$\begin{array}{ccccc} \neg a & \vee & b & \vee & c \\ b & \vee & c & \vee & d \\ a & \vee & \neg b & \vee & \neg d \end{array}$$

Figure 3.5: An example of input in Conjunctive Normal Form (CNF).

Example 3.2.6. Given the encoding in Figure 3.5, the following interpretation would be a satisfying assignment.

$$\begin{array}{rcl} a & = & \texttt{true} \\ b & = & \texttt{true} \\ c & = & \texttt{false} \\ d & = & \texttt{false} \end{array}$$

The first and second clause are satisfied by $b = \texttt{true}$. The third clause is satisfied by $a = \texttt{true}$ and also by $d = \texttt{false}$.

SAT solvers are generally also Conflict-Driven Clause Learning (CDCL) solvers. This means that the SAT solver goes through a *decide-propagate-backtrack* loop. In this mechanism, the *decide*-step the solver chooses an assignment for a previously unassigned variable. For this choice of assignment, different strategies are possible. These strategies are called *heuristics*. The most widely accepted heuristic is Variable State Independent Decaying Sum (VSIDS) [47, 70].

After the *decide*-step comes the *propagation*-step, where the solver attempts to derive additional assignments that are logical consequences of the current assignment. The propagation is often nothing more than *unit propagation*, where one derives additional assignments by checking which clauses have all but one literal left that may satisfy it. This final literal is then propagated and added to the assignment, since it is needed to satisfy the clause and no other options are available. The check for such clauses is often implemented using a two-watched-literals scheme [67]. In this scheme, every clause keeps track of two "potential" literals (also called *watches*) that may still satisfy it. If one of these watches becomes invalidated, a search for a new potential literal starts. If no new potential literal can be found the remaining watch is propagated to be `true`.

These two steps are performed in a loop until a either **(1)** the assignment provides a mapping for all variables, in which case the solver reports the model, or **(2)** a conflict in the assignment is detected and the solver performs a *backtrack*-step. In this *backtrack*-step a *learned clause* is generated based on the conflicting assignment. This learned clause is stored in a set of learned

clauses, in what is called a *learned clause database*. This learned clause prevents the solver from entering parts of the search space that resulted in this conflict. Due to the large number of learned clauses, the learned clause database is often periodically purged of learned clauses that are not useful anymore according to some metric [12].

Extending a SAT solver with Definition support

Mariën et al. [65] provide an extension of the SAT problem to one that allows the incorporation of Inductive Definitions (ID). The name of this extension is SAT(ID). A SAT(ID) solver provides a different kind of propagation: *unfounded set* (UFS). This propagation checks when there is no possible (external) cause left to fire a rule.

Example 3.2.7. As an example, consider the definition

$$\left\{ \begin{array}{l} a \leftarrow b \wedge c. \\ b \leftarrow a. \end{array} \right\}$$

and the assignment $c = \texttt{false}$. In this situation, the only way to "fire" a using the first rule is using b. But since b itself depends on a in the second rule, this is not a valid reason to make a true. Thus, in this case $\{a, b\}$ is called an unfounded set and both variables are propagated to be `false`.

In Chapter 6 we give **(1)** a description of a input format called Definition Normal Form (DEFNF) for SAT(ID) , **(2)** a more detailed explanation of the semantics used in SAT(ID), and **(3)** new selection criterion for decisions in SAT(ID).

Extending a SAT solver with Constraint Programming Techniques

Recent work has focused on extending SAT solvers (or ASP solvers) with techniques from the field of Constraint Programming (CP) [8, 21, 69]. An important part of this integration is a technique called "lazy clause generation", where, instead of eagerly, the constraints on the CP variables present in the input encoding are reduced to CNF on an on-demand basis.

Lazy Grounding

As a final extension, several techniques [18, 22, 41] aim to remove the restriction that the entire variable-free encoding must be available to the solver beforehand.

In other words, such systems would be able to perform a partial grounding step before the solving process is started. During the solving process additional parts are grounded on a by-need basis. For a detailed explanation of the lazy grounding approach that has been implemented in IDP3 we refer to the work of De Cat et al. [22].

3.3 Conclusion

We identified the KBS paradigm as an alternative way to develop software that focuses on reusability of a clear and intuitive representation of the domain knowledge associated with the problem. The strengths and weaknesses of this approach have been listed as they are currently known. The state of the IDP system before the work in the remainder of this text is explained on a high level, with references to the publications that present the techniques in more detail.

For more information, we refer the reader to a more detailed description of the system by Bruynooghe et al. [16]. This work contains extra use cases of the IDP system, as well as a more elaborated explanation and non-formal explanation of the FO($\cdot$) language with many examples.

Chapter 4

Experimental Evaluation of a State-of-the-art Grounder

This chapter is a presentation of two experiments that were published in the *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* [53].

The first experiment is described in Section 4.1 and investigates the practical behaviour of existing grounding techniques. This experiment intends to rigorously examine the run-time advantage of each of those techniques, as well as combinations of them. Using this examination we hope to better understand why these techniques are as popular as they are, as well as present a clear cost/benefit analysis to a future generation of developers of grounders. For this experiment we investigate the three grounding techniques as they have been described in Section 3.2.1. The first section of experiments also takes into account the implementation notes that have been provided in Section 3.2.1 to formulate a conclusion on the cost-to-benefit ratio of these techniques.

The second experiment is described in Section 4.2 and investigates the following question. Let $\mathcal{T}$ be some FO$(\cdot)$ theory and $\mathcal{T}_1$, $\mathcal{T}_2$ two ground theories produced by grounding $\mathcal{T}$. Also, let $\mathcal{T}_1$ be obtained by a more optimized grounding algorithm than $\mathcal{T}_2$, so that the size of $\mathcal{T}_1$ is smaller than the size of $\mathcal{T}_2$. How difficult is it for a state-of-the-art CDCL solver to find a model starting from $\mathcal{T}_1$ compared to finding a model for $\mathcal{T}_2$?

This hypothesis was initially considered after reading the work of Vaezipoor et al. [80] where there was some mention of "autarkies" that may result in a

significant increase in solving time. Although the hypothesis is never explicitly mentioned by Vaezipoor et al. [80], it was considered to be important enough to warrant a thorough examination.

4.1 Grounding Technique Experiments

In this section, we present the detailed results of the experiments we performed on the grounding techniques presented in Section 3.2.1. Table 4.1 shows the problems used during these experiments and their origins. These are all problems of the three previous ASP competitions that are classified in the category of NP-complete problems. We also added three problems that were previously used in grounding experiments performed by another group [80]. For each of these problems the experiment is run with ten instances that were randomly selected. All experiments are performed with a memory threshold of 4GB and a time threshold of 300 seconds.

We aim to investigate the effect of the three grounding techniques presented in Section 3.2.1 on the "efficiency" of the grounding step. These grounding techniques are "reduced grounding" (RED), "lifted unit propagation" (LUP), and "grounding with bounds" (GWB). In order to determine the effect on the efficiency, we compare different combinations of these techniques and measure the effect on the efficiency of the grounding phase. Efficiency is measured using three properties:

- The **number** of instances that were successfully grounded within the thresholds.
- The **duration** of the grounding phase, measured in seconds.
- The **size** of the resulting grounding, as defined in Section 3.2.1

The three identified grounding optimization techniques can be activated or disabled, so there are eight ways to combine them. The column labeled "ID" of Table 4.2 accords an identifier to each of the combinations. E.g., $\texttt{Run}_{LGx}$ represents the run where LUP and GWB were activated, and RED was not. We call any of the investigated combinations of grounding techniques "run configurations" in the remainder of this section.

In Section 4.1.1 we investigate the effect of RED by comparing $\texttt{Run}_{xxx}$ with $\texttt{Run}_{xxR}$. This allows us to determine the benefits of adding the RED technique when no other technique is used. Section 4.1.2 compares $\texttt{Run}_{xxR}$ with $\texttt{Run}_{LxR}$ to see what added benefit LUP offers when RED is used to take advantage of

Nr	Problem Name	Origin
1	15 Puzzle	ASP09
2	Blocked NQueens	ASP09
3	Channel Routing	ASP09
4	Connected Dominating Set	ASP09
5	Edge Matching	ASP09
6	Graph Partitioning	ASP09
7	Hamiltonian Path	ASP09
8	Hierarchical Clustering	ASP09
9	Maze Generation	ASP09
10	Schur Numbers	ASP09
11	Travelling Salesperson	ASP09
12	Weight Bounded Dominating Set	ASP09
13	Wire Routing	ASP09
14	Generalized Slitherlink	ASP11
15	Fastfood Optimality Check	ASP11
16	Sokoban Decision	ASP11
17	Knight Tour	ASP11
18	Disjunctive Scheduling	ASP11
19	Packing Problem	ASP11
20	Labyrinth	ASP11
21	Numberlink	ASP11
22	Reverse Folding	ASP11
23	Hanoi Tower	ASP11
24	Magic Square Sets	ASP11
25	Airport Pickup	ASP11
26	Partner Units	ASP11
27	Maze Generation	ASP11
28	Tangram	ASP11
29	Permutation Pattern Matching	ASP13
30	Graceful Graphs	ASP13
31	Bottle Filling Problem	ASP13
32	NoMystery	ASP13
33	Sokoban	ASP13
34	Ricochet Robot	ASP13
35	Solitaire	ASP13
36	Weighted Sequence Problem	ASP13
37	Incremental Scheduling	ASP13
38	Visit all	ASP13
39	Knight Tour With Holes	ASP13
40	Graph Colouring	ASP13
41	Bounded Spanning Tree	-
42	Latin Squares	-
43	Sudoku	-

Table 4.1: Problems in our benchmark set

ID	LUP	GWB	RED
$\texttt{Run}_{LGR}$	yes	yes	yes
$\texttt{Run}_{LGx}$	yes	yes	no
$\texttt{Run}_{LxR}$	yes	no	yes
$\texttt{Run}_{Lxx}$	yes	no	no
$\texttt{Run}_{xGR}$	no	yes	yes
$\texttt{Run}_{xGx}$	no	yes	no
$\texttt{Run}_{xxR}$	no	no	yes
$\texttt{Run}_{xxx}$	no	no	no

Table 4.2: Different experiment setups

the extra information derived by LUP. Section 4.1.3 contains a report of three comparisons to examine the advantages that GWB offers.

Detailed information for all run configurations is presented in Table 4.3. Since $\texttt{Run}_{Lxx}$ and $\texttt{Run}_{xGx}$ are not used in any of the comparisons above, no experiments are performed for these combinations. For each run configuration (identified in the leftmost column) we present three statistics:

- $s_{\#}$ is the number of instances that were successfully grounded,
- t_{avg} is the average running time of the successfully grounded instances, and
- g_{avg} is the average grounding size of the successfully grounded instances.

Table 4.3 identifies the run configurations for which the highest number of successfully grounded instances was produced by showing these in **bold**. The last row of the table also shows the total number of successfully ground instances for each run configuration. When not a single instance was successfully grounded, the table shows an "-" for average time running and grounding size.

Examining Table 4.3, we observe the following.

- The three run configurations that have the most successfully grounded instances are the three run configurations including GWB ($\texttt{Run}_{LGR}$, $\texttt{Run}_{LGx}$, and $\texttt{Run}_{xGR}$).
- There is only one problem (37) for which none of the approaches could successfully ground even a single instance. This problem is known to favor the usage of Constraint Programming (CP) techniques. IDP3 offers an option in which these techniques are used, but this option was not turned on for this problem in our benchmark set.

- There is one problem (36) for which none of the approaches have an effect on the efficiency of the grounding phase. This can also be explained by the fact that this is a problem for which CP does very well. In contrast to problem 37, IDP3's CP option was turned on for this problem.
- The table shows that there is variety in the difficulty; some problems can be ground by the naive approach and other problems require techniques.
- Consecutive ASP competitions became harder to ground. ASP09 problems can be ground by any run configuration. ASP11 contains a few problems for which the most naive run configuration could not ground a single instance. ASP13 has problems for which multiple approaches could not ground a single instance.
- Generally, when the resulting grounding size is smaller, the grounding time is also reduced.

All comparisons are presented in Table 4.4. For the comparison between $\texttt{Run}_i$ and $\texttt{Run}_{i'}$ (identified in the first column) we present two statistics:

- $t^{\%}_{avg}$ is the ratio (in percent) of the average running time of $\texttt{Run}_{i'}$ over the average running time of $\texttt{Run}_i$ when both approaches succeeded. I.e., t_{avg} of $\texttt{Run}_{i'}$ divided by t_{avg} of $\texttt{Run}_i$, only counting instances where both $\texttt{Run}_i$ and $\texttt{Run}_{i'}$ succeed.
- $g^{\%}_{avg}$ is the ratio (in percent) of the average grounding size of $\texttt{Run}_{i'}$ over the average grounding size of $\texttt{Run}_i$ when both approaches succeeded. I.e., g_{avg} of $\texttt{Run}_{i'}$ divided by g_{avg} of $\texttt{Run}_i$, only counting instances where both $\texttt{Run}_i$ and $\texttt{Run}_{i'}$ succeed.

It is important to note that comparisons are only made between instances that both approaches could successfully ground. As a result, the ratios presented in Table 4.4 cannot be obtained by dividing the corresponding values present in Table 4.3, unless ofcourse, both approaches were able to solve the exact same set of instances. When not a single instance was successfully grounded by both approaches, the table shows an "-" for ratio of average time and grounding size.

4.1.1 Experimental Evaluation of RED

The comparison between $\texttt{Run}_{xxx}$ and $\texttt{Run}_{xxR}$ is shown in the first row of Table 4.4. This comparison shows that adding RED is very benificial to the efficiency of the grounding step; on average the grounding is done in 71.66% of the time and

ID	Run_{LGR}			Run_{LGx}			Run_{LxR}			Run_{xGR}			Run_{xxR}			Run_{xxx}		
	$s_{\#}$	t_{avg}	g_{avg}	$s_{\#}$	t_{avg}	g_{avg}	$s_{\#}$	t_{avg}	g_{avg}	$s_{\#}$	t_{avg}	g_{avg}	$s_{\#}$	t_{avg}	g_{avg}	$s_{\#}$	t_{avg}	g_{avg}
1	**10**	0	215558	**10**	0	223240	**10**	0	215558	**10**	0	222583	**10**	0	222583	**10**	0	296408
2	**10**	1	3190	**10**	1	13312	**10**	14	3190	**10**	1	4992	**10**	14	4992	9	45	17904694
3	**10**	3	74836	**10**	3	126229	8	86	49209	10	3	74836	8	85	49209	4	131	117658700
4	**10**	0	6952	**10**	0	59696	**10**	0	6952	**10**	0	6952	**10**	0	6952	**10**	0	161406
5	**10**	4	4116943	**10**	4	6478315	**10**	13	4116943	**10**	3	4116943	**10**	12	4116943	7	16	42750706
6	**10**	0	34301	**10**	0	194975	**10**	0	34301	**10**	0	34301	**10**	0	34301	**10**	0	194975
7	**10**	0	2600	**10**	0	4549	**10**	0	2600	**10**	0	78234	**10**	0	78234	**10**	0	141799
8	**10**	1	48132	**10**	1	68761	9	0	47722	9	1	776837	9	1	776837	9	1	1225968
9	**10**	0	9692	**10**	0	45014	**10**	1	9692	**10**	0	11934	**10**	1	11934	**10**	3	3186601
10	**10**	0	71502	**10**	0	71537	**10**	0	73406	**10**	0	73144	**10**	0	73144	**10**	0	75076
11	**10**	0	16880	**10**	0	47877	**10**	0	30782	**10**	0	16604	**10**	0	16604	**10**	1	1366231
12	**10**	0	1100	**10**	0	29281	**10**	1	1100	**10**	0	1100	**10**	1	1100	**10**	8	10176834
13	**10**	0	112314	**10**	0	309399	**10**	25	112314	**10**	0	125686	**10**	25	125686	6	31	48805667
14	**10**	9	1063733	7	6	9049236	5	100	154492	5	47	33537369	5	139	33537369	0	-	-
15	**10**	1	15903	**10**	2	1013460	**10**	20	15903	**10**	16	14349222	**10**	39	14349222	0	-	-
16	**10**	1	1507466	**10**	4	10195036	**10**	12	1507466	4	37	73869847	4	38	73869847	0	-	-
17	**10**	2	30806	**10**	2	50625	**10**	15	30806	8	7	2293888	8	9	2293888	7	4	1586793
18	**3**	30	1904374	**3**	30	2089858	1	23	1737497	**3**	30	1904374	1	22	1737497	0	-	-
19	**3**	11	6635640	2	6	3969862	**3**	11	6635640	2	6	3969748	2	6	3969748	2	6	3970001
20	**8**	72	632396	3	33	45063936	**8**	90	632396	6	37	6059844	6	40	6059844	3	37	47697810
21	**10**	0	150285	**10**	0	406347	7	37	14247	**10**	0	151076	7	37	14615	2	10	26014222
22	**1**	10	1487006	**1**	15	5509897	**1**	31	1487006	**1**	10	1492710	**1**	31	1492710	**1**	63	22641079
23	**10**	1	884547	**10**	3	5667748	**10**	1	884547	**10**	9	16027486	**10**	9	16027486	**10**	12	19663568
24	**10**	0	2804	**10**	0	83751	**10**	0	2804	**10**	0	6146	**10**	0	6146	7	3	102586
25	**8**	25	25194237	**8**	25	32407583	2	66	3348329	**8**	24	25328233	2	65	3368745	0	-	-
26	**10**	1	3813188	**10**	1	5240087	**10**	20	3813188	**10**	1	3813388	**10**	21	3813388	0	-	-
27	**10**	65	50060	**10**	66	815970	4	87	27996	**10**	65	202036	2	60	91321	0	-	-
28	**10**	16	367124	0	-	-	**10**	34	367124	**10**	16	367124	**10**	34	367124	0	-	-
29	**10**	28	4458517	**10**	29	4481669	5	34	529380	**10**	29	4458517	5	33	529380	4	30	1601321
30	**10**	0	31059	**10**	0	33536	**10**	0	31059	**10**	0	31059	**10**	0	31059	**10**	0	36893
31	**10**	5	1045989	**10**	6	4029795	0	-	-	**10**	4	1063372	0	-	-	0	-	-
32	**5**	13	6735660	4	17	18976800	0	-	-	2	27	16920003	0	-	-	0	-	-
33	**10**	5	2113681	**10**	5	3082032	3	77	133749	**10**	5	2113834	3	77	133801	0	-	-
34	**10**	26	7705908	**10**	27	10077373	**10**	41	7705908	**10**	35	16966275	**10**	49	16966275	**10**	51	19420756
35	**10**	0	35550	**10**	0	882915	**10**	4	35550	**10**	0	43156	**10**	10	43156	0	-	-
36	**10**	0	1702	**10**	0	1702	**10**	0	1702	**10**	0	1702	**10**	0	1702	**10**	0	1702
37	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-
38	**10**	0	25715	**10**	0	485210	**10**	0	25715	**10**	0	314532	**10**	0	314532	**10**	0	957380
39	**10**	13	2400376	**10**	14	3189129	0	-	-	0	-	-	0	-	-	0	-	-
40	**10**	0	15535	**10**	0	32013	**10**	0	15535	**10**	0	15535	**10**	0	15535	**10**	0	162686
41	**10**	0	46834	**10**	0	50334	**10**	0	46834	**10**	0	277584	**10**	0	277584	**10**	0	281084
42	**10**	1	1888556	**10**	1	2775305	**10**	2	1888556	**10**	3	4779401	**10**	3	4779401	**10**	3	4860001
43	**10**	5	1109217	**10**	5	1547641	**10**	80	1194156	**10**	5	1183534	**10**	76	1183534	5	4	2181169
Sum	**388**			368			326			358			313			226		

Table 4.3: Results for all discussed combinations of grounding techniques

the resulting grounding is only 37.62% as big. Moreover, for some problems, the decrease in ground size is several orders of magnitude (e.g. problems 2, 3, and 21). On the other hand, there are problems for which this technique has no effect (e.g. problems 10, 19, and 36).
Table 4.3 shows that the addition of RED also has a positive influence on the number of successfully grounded instances, going from 226 to 313. From this we can conclude that the addition of RED without any additional techniques has positive effects on the efficiency of the grounding and has no drawbacks. Combined with the fact that the implementation of this technique is rather straightforward, as mentioned in Section 3.2.1, it is highly advisable to implement it.

4.1.2 Experimental Evaluation of LUP

This section compares $\texttt{Run}_{xxR}$ with $\texttt{Run}_{LxR}$ to see what added benefit LUP offers when RED is used to take advantage of the extra information derived by LUP. The experimental data for the comparison can be found in Table 4.4 in the second row. This comparison shows that the additional information that LUP derives can improve the usage of the RED technique even further; on average the grounding is done in 87.62% of the time and the resulting grounding is only 69.03% as big. Additionally, the number of successfully grounded instances increases from 313 to 326. For some problems no additional information could be derived, so the grounding size remains the same, whilst increasing the average running time (e.g. problems 3, 5, and 18). From this we can derive that the computational cost of the LUP execution is acceptable; even for problems in which it derives nothing and thus offers no advantages, the average grounding time increases at most by 5%. For LUP we can thus conclude that the implementation of this technique is definitely worthwhile. Whilst it offers no improvements in the efficiency of the grounding step as a standalone option when no other options are activated, the extra elements it derives can be exploited by other grounding techniques. We also note that the implementation of this technique is more challenging than the implementation of RED, since special data structures and reasoning techniques need to be implemented to create and query the symbolic representation. Even with our approximative implemention of LUP (as mentioned in Section 3.2.1), we observe an increased average grounding time for some of the problems. This indicates special care needs to be taken to not make the implementation too costly.

4.1.3 Experimental Evaluation of GWB

In order to determine the effect of the GWB technique we perform three comparisons. First we compare $\texttt{Run}_{xxR}$ with $\texttt{Run}_{xGR}$ to determine the benefit of using GWB in combination with RED as opposed to when only RED is used. Next we compare $\texttt{Run}_{LxR}$ with $\texttt{Run}_{LGR}$ to see analyse how much GWB benefits from the extra information derived by LUP. To illustrate the approximative nature of our GWB implementation, we compare $\texttt{Run}_{LGx}$ with $\texttt{Run}_{LGR}$. This will give an idea of how much derivations were "missed" because of approximative method.

The comparison between $\texttt{Run}_{xxR}$ and $\texttt{Run}_{xGR}$ is shown in the third row of Table 4.4 and it shows that on average the grounding is done in 62.58% of the time. As argued when introducing the GWB technique (see Section 3.2.1), results show that the grounding size remains the same. This shows that although GWB and RED have a similar effect (i.e., reduce the grounding size), the advantage that the GWB technique does this *beforehand* leads to a substantial decrease in grounding time.

The fourth row of Table 4.4 shows a similar comparison of $\texttt{Run}_{LxR}$ with $\texttt{Run}_{LGR}$. The effect of adding GWB when both LUP and RED are already activated leads to a grounding time that is on average 54.67% as long as without GWB. This is the largest decrease in grounding time for the addition of a single technique. This additionally shows that GWB is able to use the extra information derived by LUP effectively; it reduces the grounding time to 54.67% of the former time, as opposed to the 62.58% reduction that was witnessed in the previous comparison in which LUP was absent. The grounding size is reduced as well (98.68%). This is counter-intuitive because we argued in the previous section that adding GWB when RED is already present the grounding size should remain the same. This is also confirmed by the third row in the table. We examined this and discovered that this reduction in grounding size is caused by three problems where there is a certain lack of optimization in the implementation of our technique. More specifically, some Tseitin symbols are introduced twice where only one was necessary for $\texttt{Run}_{LxR}$. On the other hand, $\texttt{Run}_{LGR}$ did not contain these duplicates.

The comparison between $\texttt{Run}_{LGx}$ and $\texttt{Run}_{LGR}$ is shown in the last row of Table 4.4. The grounding size difference here shows the approximative nature of our GWB technique. If GWB was complete, the addition of the RED technique would not lead to a smaller average grounding size. This is constrasted by the observation that in this comparison, the average grounding size is reduced to 47.24% of the original size. This means that the simplifications that are not done by GWB due to its approximative nature account for over half of the remaining

Comparison	$t^{\%}_{avg}$	$g^{\%}_{avg}$
$\texttt{Run}_{xxx}$ vs $\texttt{Run}_{xxR}$	71.66	37.62
$\texttt{Run}_{xxR}$ vs $\texttt{Run}_{LxR}$	87.62	69.03
$\texttt{Run}_{xxR}$ vs $\texttt{Run}_{xGR}$	62.58	100
$\texttt{Run}_{LxR}$ vs $\texttt{Run}_{LGR}$	54.67	98.68
$\texttt{Run}_{LGx}$ vs $\texttt{Run}_{LGR}$	85.94	47.24

Table 4.4: Ratios of average grounding time and size between runs

grounding size. The significant speedup offered by GWB, combined with the fact that the implementation is highly approximative (on average half of the remaining grounding size could be prevented by GWB but had to be simplified by RED) serves as a good motivation towards future work investigating the possibility to reduce the approximative nature of the GWB method without incurring too much overhead.

As was the case with LUP, the implementation of the GWB technique is challenging. Nonetheless, we observe that this technique is essential when building a state-of-the-art grounder. Despite its current approximative implementation in IDP3, GWB appears to be the best of the three discussed techniques to decrease the grounding time.

4.2 Solver Behaviour on Optimized Ground Theories

As mentioned in our introduction, the IDP3 system uses the ground-and-solve approach. The previous section was concerned with investigating the effects of RED, LUP, and GWB on the grounding step. This section is dedicated to examining the effect that a smaller grounding has on the search process. We state this question more clearly.

Let $\mathcal{T}$ be some FO$(\cdot)$ theory and $\mathcal{T}_1$, $\mathcal{T}_2$ two ground theories produced by grounding $\mathcal{T}$. Also, let $\mathcal{T}_1$ be obtained by a more optimized grounding algorithm than $\mathcal{T}_2$, so that the size of $\mathcal{T}_1$ is smaller than the size of $\mathcal{T}_2$. How difficult is it for a state-of-the-art CDCL solver to find a model starting from $\mathcal{T}_1$ compared to finding a model for $\mathcal{T}_2$?

Because of the overhead associated with larger groundings, one expects that it is harder to find a model for a large ground theory than it is to find for a smaller one. Thus it is expected to take less time to find a model M_1 for $\mathcal{T}_1$ than to find a model M_2 for $\mathcal{T}_2$. This is experimentally confirmed by Vaezipoor et al. [80] on

the Bounded Spanning Tree, Latin Squares, and Sudoku problem set. In this section, we investigate in more detail what aspect of finding a model becomes harder: is the actual search tree for finding M_2 bigger than that for finding M_1? Or is the cause of the slowdown simply due to the overhead associated with a larger grounding?

We first present a simple example to show what such a difference between $\mathcal{T}_1$ and $\mathcal{T}_2$ might look like. Assume

$$\mathcal{T} = \varphi \wedge P \wedge (P \vee (\psi \wedge \pi))$$

A simple grounding mechanism would introduce a Tseitin literal [78] L_T to unnest the rightmost conjunct, which results in

$$\mathcal{T}_2 = \varphi \wedge P \wedge (P \vee L_T) \wedge (L_T \Leftrightarrow \psi \wedge \pi)$$

While a smart grounding algorithm using LUP could derive that P must be true, and hence could use RED to obtain a simpler, flattened theory

$$\mathcal{T}_1 = \varphi \wedge P$$

So the difference between $\mathcal{T}_1$ and $\mathcal{T}_2$ is the fact that the constraint $(P \vee (\psi \wedge \pi))$ is no longer present in $\mathcal{T}_1$, since it is a logical consequence of $\mathcal{T}_1$. As observed by Vaezipoor et al. [80], the unit propagation (UP) capability of a CDCL solver does not eliminate the difference between optimized groundings. For example, UP should also derive P to be true, but the resulting theories are still not equal in size:

$$UP(\mathcal{T}_1) = \varphi$$

and

$$UP(\mathcal{T}_2) = \varphi \wedge (L_T \Leftrightarrow \psi \wedge \pi)$$

So solving $\mathcal{T}_1$ will be "easier" than solving $\mathcal{T}_2$, even with unit propagation of P taken into account. However, it is theoretically easy to find some assignment satisfying $(L_T \Leftrightarrow \psi \wedge \pi)$: simply assign L_T the truth value of $\psi \wedge \pi$, which is always possible since L_T is a Tseitin variable not occurring in φ. On the other hand, a solver might choose a value for L_T before the truth value of $(\psi \wedge \pi)$ is known, potentially incurring an exponential blowup of the search space. In this section, we investigate how much trouble a modern solver has with the extra constraints in an unoptimized ground theory.

In theory, using an optimized small grounding instead of a naive, larger one can speed up the search process in two aspects:

(1) The solver has to keep track of fewer things, smaller formulas, and assign values to fewer variables, typically reducing the solving time by a factor proportional to the reduction of the size of the ground theory.

(2) The omitted ground formulas represent hard constraints for the solver, so the optimized grounding represents a less complex problem. As a result, the search tree needed to find a model is smaller, potentially leading to an exponential speedup.

To decide which aspect is the dominant one, we extract from the benchmark set introduced in the previous section those instances that could be grounded by both the most naive grounding approach ($\texttt{Run}_{xxx}$) and the most advanced one ($\texttt{Run}_{LGR}$) in under 300 seconds. This resulted in a set of 226 instances, for which a distribution of the grounding sizes is given in Figure 4.1. It is clear that, on average, the resulting sizes of the optimized groundings are more than ten times smaller than the sizes of the naive groundings.

On these two ground theories we run MiniSAT(ID), IDP3's state-of-the-art CDCL-based solver, for each instance and compare the results. For this ground-and-solve workflow IDP3 was given a 900 second time limit, as well as a 4GB memory limit. The time needed for MiniSAT(ID) to solve the grounded instances is given in Figure 4.2. A first conclusion is that out of the 226 instances, 32 more could be solved when using the optimized grounding, and that, on average, it takes more time to solve $\texttt{Run}_{xxx}$ instances than $\texttt{Run}_{LGR}$ instances. These findings are consistent with earlier experiments [80] and the general intuition that smaller groundings lead to reduced solving time.

Note that CDCL solvers such as MiniSAT(ID) follow a depth-first search strategy: at every search step, the solver *decides* a new atom to assign a truth value to, *propagates* implied truth values of other literals, and checks whether the assignment of truth values is still consistent. If not, a *conflict* occurs, and the CDCL solver backjumps to a consistent state. To verify the cause of the solving time difference for optimized and unoptimized groundings, we plot the number of decisions, propagations, and conflicts of our solve runs in Figures 4.3, 4.4, and 4.5 respectively. From these three measures, the number of conflicts is the best representative for search tree width, since every conflict results in a backjump step, triggering the exploration of a new branch in the search tree. The number of decisions often also is an indicator of search tree width, but a high number of decisions does not necessarily entail a high number of branches in a search tree. For instance, an unconstrained problem with n variables will require n decisions to be made, even though the search tree never branches. The number of propagations on the other hand is a measure of overhead: if two solving runs incur the same number of conflicts, but a different number of

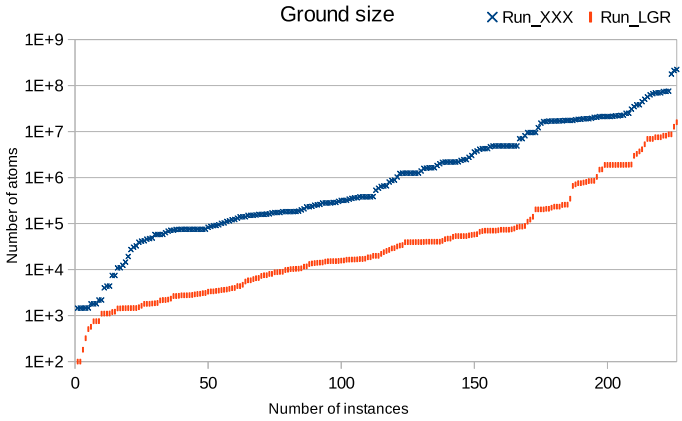


Figure 4.1: Cactus plot of ground sizes for instances grounded by $\texttt{Run}_{xxx}$ and $\texttt{Run}_{LGR}$.

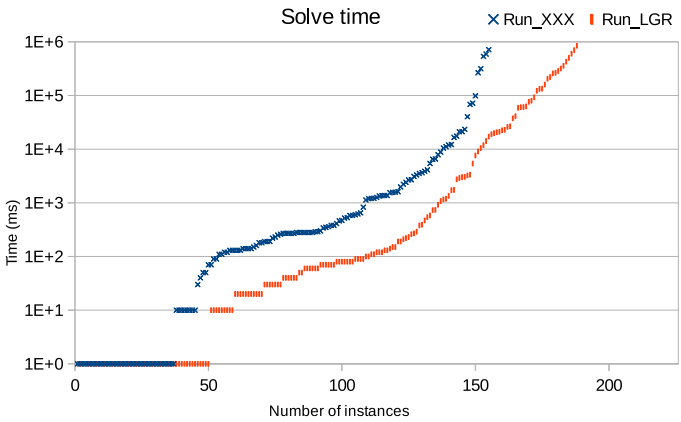


Figure 4.2: Cactus plot of solving times for instances grounded by $\texttt{Run}_{xxx}$ and $\texttt{Run}_{LGR}$.

propagations, then the solving run with the most propagations will have done more work in each branch of the search tree, and will typically take more time.

So from a theoretical point of view, problems for which a solver obtained many conflicts are hard for that solver, whilst problems with many propagations in each search branch simply state that many variables had to be assigned a value to keep the solver state consistent, implying that there are either a lot of variables, or a lot of constraints to keep track of.

Given these thoughts, we see in Figure 4.5 that the amount of instances solved with relatively few conflicts (less than ten thousand) is equal between both grounding approaches. It is only when the number of conflicts gets high that optimizing the grounding leads to more solved problem instances. A similar observation can be made in Figure 4.3 for the number of decisions of each solving run. Figure 4.4 on the other hand shows is different because even for problems with relatively few propagations the optimized grounding instance requires significantly less propagations than its unoptimized counterpart.

These observations are consistent with the hypothesis that the most significant cause for solving time speedup with optimized groundings is simply the reduction of overhead incurred by, e.g., performing more propagations during solving. The above observations are *not* consistent with the hypothesis that the most significant cause for solving time speedup with optimized groundings is the reduction of the search tree inferred by the absence of constraints. Using this hypothesis, we would expect the number of conflicts for solver runs on the unoptimized instances to be significantly larger than the number of conflicts for solver runs on the optimized instances, over the whole range of instances. The increase observed between runs with more than ten thousand conflicts can be explained by the fact that faster solving times for the optimized grounding (see Figure 4.2) lead to more plot points. These extra plot points will generally be associated with a large number of conflicts, since they represent difficult search problems that resulted in a solving timeout for the naive grounding.

To further investigate this issue, we conducted a statistical analysis of our data. We are trying to support or debunk the claim that optimizing the grounding leads to a less difficult search problem, or alternatively, leads to less overhead during search. We made the reasonable assumption that the number of conflicts during search is a good indicator of problem complexity, and that the number of propagations is a good indicator of problem overhead. All that is left to check is whether a large increase in ground size due to disabling grounding optimizations leads to a large increase of conflicts or propagations when solving the problem instance. We quantify an increase in ground size of a problem instance as the ratio of the size of the unoptimized grounding to the size of the optimized grounding (i.e., grounding size of $\texttt{Run}_{xxx}$ divided by grounding

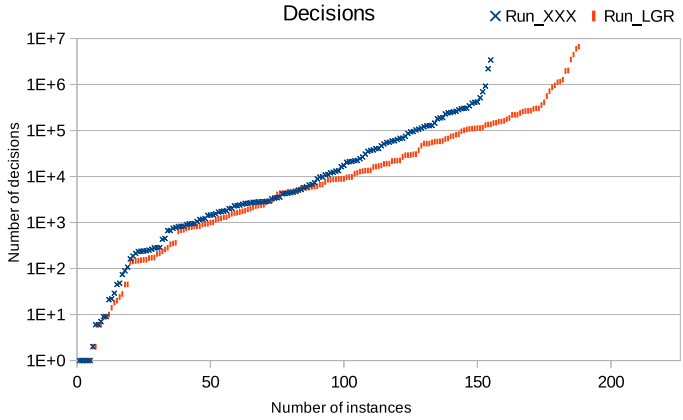


Figure 4.3: Cactus plot of the number of decisions made by MINISAT(ID) while finding a model for instances grounded by $\texttt{Run}_{xxx}$ and $\texttt{Run}_{LGR}$.

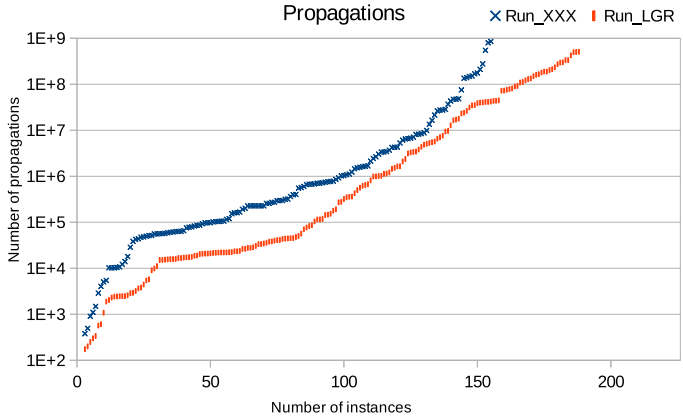


Figure 4.4: Cactus plot of the number of literals propagated while finding a model for instances grounded by $\texttt{Run}_{xxx}$ and $\texttt{Run}_{LGR}$.

size of $\texttt{Run}_{LGR}$). Similarly, an increase in conflicts (propagations) is quantified as the ratio of conflicts (propagations) observed when solving the unoptimized grounding to the number of conflicts (propagations) observed when solving the optimized grounding. This second quantification is only meaningful when the solving run did not hit the timeout, so we restrict our benchmark set to the 160 instances for which both the optimized and unoptimized grounding were solved.

Given these formal notions, we now check the correlation between increases in

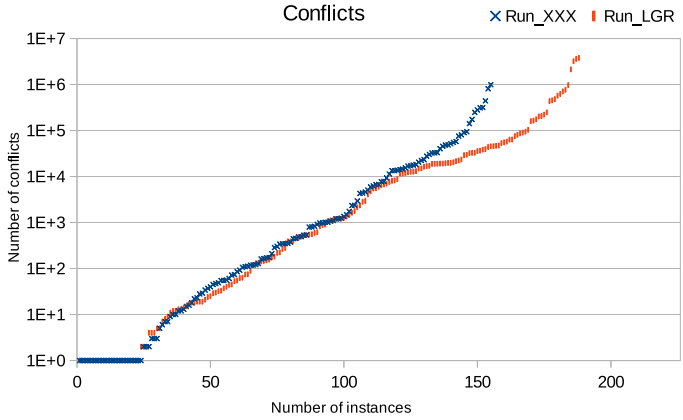


Figure 4.5: Cactus plot of encountered conflicts while finding a model for instances grounded by $\texttt{Run}_{xxx}$ and $\texttt{Run}_{LGR}$.

correlated variables	Spearman ρ	95% confidence interval
↑ ground size, ↑ conflicts	0.046	$[-0.109, 0.198]$
↑ ground size, ↑ propagations	0.617	$[0.511, 0.704]$

Table 4.5: Correlating increases in ground size.

ground size to increases in conflicts or increases in propagations. This is done by calculating Spearman's rank correlation coefficient for both of these measures. The results, shown in Table 4.5, indicate that an increase in ground size is only marginally associated with an increase in the number of conflicts, while an increase in ground size is strongly correlated to an increase in propagations. Also, the obtained correlations are well within appropriate error margins.

Given this statistical data, and given the preceding plot-based observations, we find that optimizing the grounding process does **not** significantly reduce the search tree of a subsequent solving step, but it does lead to less overhead for the solver during search. This can intuitively be explained by the fact that it is relatively simple to derive that these omissible constraints are logical consequences of the original theory, since we were able to detect during grounding that the constraints could be omitted. Nonetheless, it goes to the credit of modern CDCL solvers that they are not distracted by these extra constraints, but instead seem to largely ignore them in their search trees. We suspect that activity-based heuristics, which prioritize assignments to variables occurring in difficult constraints, play a key role in the observed behaviour.

4.3 Conclusion

This chapter contained two experiments that investigated the impact of the usage of the grounding techniques presented in Section 3.2.1 on **(1)** the efficiency of the grounding phase and **(2)** the behaviour of the solving phase.

The first experiment aims to identify the cost-benefit ratio of the presented grounding techniques. The efficiency of different combinations of uses of the grounding techniques "reduced grounding" (RED), "lifted unit propagation" (LUP), and "grounding with bounds" (GWB) are compared.

The conclusions are that RED is an easy to implement technique that offers significant benefits to both the grounding size and the grounding time. We identified it as an essential element in any modern grounder. The experiments show that the LUP technique offers advantages in the sense that it derives extra information that can be exploited by other techniques. Using this technique allows, e.g., RED to further reduce the grounding size. Care must be taken to ensure that the overhead of LUP does not become too great. This is especially important taking into account the fact that this technique requires a complex implementation of a symbolic representation and a query engine for this representation. The GWB technique relies on the same symbolic representation but instead uses it to derive quantifier instances that do not have to be considered during the grounding phase. This technique could be seen as an approximate derivation of what eliminations the RED technique is able to do. The addition of this technique causes the highest speedup that was witnessed in the experiment set. This is because this technique detects obsolete quantifier instances *before* they are processed. As was shown by the experiments, this technique is approximate and not a replacement for RED. The GWB shares the same overhead concerns as the LUP technique since it relies on the same implementation of a symbolic representation and a querying engine for that representation.

The second set of experiments investigates the following question. Let $\mathcal{T}$ be some FO($\cdot$) theory and $\mathcal{T}_1$, $\mathcal{T}_2$ two ground theories produced by grounding $\mathcal{T}$. Also, let $\mathcal{T}_1$ be obtained by a more optimized grounding algorithm than $\mathcal{T}_2$, so that the size of $\mathcal{T}_1$ is smaller than the size of $\mathcal{T}_2$. How difficult is it for a state-of-the-art CDCL solver to find a model starting from $\mathcal{T}_1$ compared to finding a model for $\mathcal{T}_2$? In the section that discusses these experiments we give an example of how two different grounding can encode the same models. As part of this investigation, we argue that the number of propagations during the solving phase is a good indication of how much overhead the solver has, and that the number of conflicts during the solving phase is a good indication of how "hard" it was to solve the problem. After running a benchmark that

considers two ways of generating groundings that differ with about a factor of 10 in size, we compare the solving behaviour on these groundings. Our findings were illustrated by several figures. See Figure 4.4 (Figure 4.5) for the graphs that show the difference in the number of propagations (the difference in the number of conflits). Afterwards, we calculated Spearman's rank correlation coefficient to see how both of these measures are correlated with an increase in grounding size. A statistically significant correlation between the grounding size and the number of propagations for solving that grounding was found. This indicates that a larger grounding leads to more overhead during the solving phase. No statistically significant correlation between the grounding size and the number of conflicts for solving that grounding was found. This indicates that a larger grounding does **not** lead to a more complex solving process with a bigger search tree.

Chapter 5

Complementing the IDP3 system with XSB

This chapter discusses the augmentation of the IDP3 system with techniques from the Logic Programming (LP) community. The contents of this chapter are published in the proceedings of the 29th International Conference on Logic Programming (ICLP'13) [56] and the Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments (CICLOPS'14) [55].

We start this chapter with some preliminaries in Section 5.1. Section 5.1.1 analyses the structure of the IDP3 system and presents it as a sequence of subtasks that need to be performed. In Section 5.1.2 we further refine one of these subtasks into a sequence of evaluations of symbols defined in a definition for a given interpretation. Next, Section 5.2 contains a detailed specification on how to evaluate these symbols using techniques from the field of Logic Programming (LP) by providing a transformation into a Logic Program that is executable by `XSB`. Section 5.4 extends this use of `XSB` by *partially* evaluating definitions that cannot be fully evaluated yet. We conclude in Section 5.5.

5.1 Preliminaries

In this section we lay the groundwork for our transformation to a Prolog program. First we present a more detailed view of the workflow of IDP3. Then we specify

```
procedure modelexpand(T_in, I_in, Σ_in) {
    if( not (sanitycheck(T_in, I_in, Σ_in))) {
        // throw error and exit.
    }
    T_typed = infertypes(T_in, Σ_in);
    T_split = splitdefinitions(T_typed);
    T_pre, T_search, T_post, T_forget = splittheory(T_split);
    I_search = calculatedefinitions(T_pre, I_in);
    T_norm = normalize(T_search);
    T_ground = ground(T_norm, I_search);
    I_solved = solve(T_ground);
    I' = postprocess(T_post, I_solved);
    return I';
}
```

Listing 5.1: The IDP3 model expansion workflow

more formally the subtask that may be solved by querying a Prolog Program that encodes the same problem.

5.1.1 Workflow Analysis of IDP3

The IDP3 system is described in Section 3.2 as a *ground-and-solve* system. The actual workflow of the system is a little more complicated [15, 23] and does not only consist of a "ground" and a "solve" step. Several steps are added as a necessity, others for their positive impact on the efficiency (i.e., running time, memory requirements) of the system. The full list of the subtasks, in sequence, is given by Bogaerts et al. [15]. Below we present this list, with an additional first step that checks the validity[1] of the FO(·) specification. A more procedural representation is given in Listing 5.1.

(1) Sanity check: Checks the validity of Σ, $\mathcal{T}$, and I. If the given specification is not "valid" as was defined in Section 2.2, abort the model expansion task and return an error. Some common errors are (a) the given $\mathcal{T}$ or I contains symbols not declared in Σ, (b) the given I contains invalid predicate or function interpretations (e.g., a predicate interpretation that maps some tuple to `true` as well as `false`), or (c) the given $\mathcal{T}$ contains variables that are not scoped.

(2) Type inference: Derives a unique well-typed theory from the partially-typed input theory. If none or multiple exist, an error is produced and the workflow ends here.

[1] as was defined in Section 2.2

(3) Definition splitting: Split a definition into multiple definitions as much as possible.

(4) Theory splitting: Split the theory used for model expansion into four parts: (a) **Preprocess**: a part whose models can be computed efficiently in advance, (b) **Search**: a part without special properties, (c) **Postprocess**: a part that can be evaluated in a post-processing step, and (d) **Forget**: a part that is irrelevant for the task at hand and that can be ignored without changing soundness of the model expansion inference.

(5) Calculate definitions: Perform model expansion on the **Preprocess** part.

(6) Theory normalization: Transform **Search** into an equivalent theory **Search**$'$ in a suitable normal form (e.g., quantifications and negations are pushed inwards).

(7) Grounding: Ground **Search**$'$.

(8) Solving: Apply a search algorithm to the ground theory of the previous step.

(9) Postprocessing: Perform model expansion on the **Postprocess** part to complete the (partial) model found in the previous step.

In addition to this, Bogaerts et al. [15] propose to use a bootstrapping approach for steps 2, 3, 4, and 6. I.e., performing these tasks is considered a model expansion task on some meta-vocabulary that describes the original FO($\cdot$) specification.

5.1.2 The Calculating Definitions Step

In this section we define several categories for symbols occurring in $\mathcal{T}$. We then link these categories to the concept of theory splitting and specify which ones end up in $\mathcal{T}_{pre}$. As a final part, we show how the calculating definitions step on $\mathcal{T}_{pre}$ is resolved.

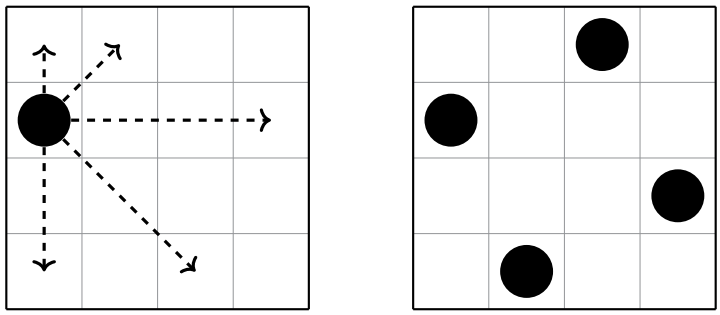
The set of *defined* symbols (denoted Σ_{def}) contains symbols in Σ that appear in a head of a rule (with a non-empty body) of a definition in $\mathcal{T}$. We define the set of *open* symbols, denoted as Σ_{open}, as $\Sigma \setminus \Sigma_{def}$, i.e., the set of symbols that occur only in the body of any of the rules in $\mathcal{T}$. The set of *input* symbols (denoted Σ_{input}) contains any symbol σ for which σ^I is two-valued, with I the input structure. Note that any input symbol is either an open symbol or a defined symbol but never both.

We define the *dependency relation* between symbols as follows. A (predicate or function) symbol σ_1 depends on (predicate or function) symbol σ_2 in theory $\mathcal{T}$ (denoted $dep_{\mathcal{T}}(\sigma_1, \sigma_2)$) if $\mathcal{T}$ contains a rule that defines σ_1 and if σ_2 appears in the body of that rule.

The set of *search* symbols (denoted Σ_{search}) is defined inductively as follows: σ is a *search* symbol if **(1)** σ is an open predicate but not an input predicate, or **(2)** if σ depends on a symbol σ' and σ' is a *search* predicate.

The set of *input∗* symbols (denoted Σ_{in*}) is defined as $\Sigma_{def} \setminus \Sigma_{search}$. These input$*$ symbols can be calculated in advance. In the ASP and the Prolog communities, the input symbols are called the *extensional* predicates and the input$*$ symbols are called the *intentional* predicates. The calculation of which symbols belong to which categories is out of the scope of this thesis. Interested readers can consult Bogaerts et al. [15] for a bootstrapping approach to do this.

Example 5.1.1. We introduce a running example for the remainder of this chapter: the *n*queens problem. The *n*queens problem consists of a chessboard of dimensions n by n. The goal is to place n queens on the board in a way such that they cannot strike one another. In chess, a queen can attack in a horizontal, vertical, and diagonal fasion. A graphical representation of the way a queen may strike and a solution to this problem is given in Figure 5.1.



(a) Queen striking directions (b) *n*queens solved for $n = 4$

Figure 5.1: Representation of striking directions of a queen and of a solved *n*queens puzzle for $n = 4$

An FO$(\cdot)$ encoding for this problem is given in Listing 5.2. The given FO$(\cdot)$ encoding for the *n*queens problem *defines* the first and second diagonal for a square at (x, y) in lines (1) and (2). Lines (3) and (4) express that there must be exactly one queen on each row and column, respectively. The restriction that at most one queen may appear on any diagonal is expressed in lines (5) and (6).

```
vocabulary Σ_nq {
  type index isa nat       // x- or y-index on the grid
  type diag isa nat        // diagonal number
  queen(index,index)       // squares that contain queens
  n:index                  // n = board size (= max index)
  diag1(index,index):diag  // 1st diagonal of a square
  diag2(index,index):diag  // 2nd diagonal of a square
}

structure I_nq : Σ_nq {
  index = {1..4}
   diag = {1..7}
      n = 4
}

theory T_nq : Σ_nq {
  {
    diag1(x,y) = d ← d = x - y + n.              // (1)
    diag2(x,y) = d ← d = x + y - 1.              // (2)
  }

  ∀ x: ∃=1 y : queen(x,y).                        // (3)
  ∀ y: ∃=1 x : queen(x,y).                        // (4)

  ∀ d: #{x y : queen(x,y) ∧ diag1(x,y) = d} < 2. // (5)
  ∀ d: #{x y : queen(x,y) ∧ diag2(x,y) = d} < 2. // (6)
}
```

Listing 5.2: An FO(·) specification for $n$queens

For this example, the categories of symbols are as follows.

- $\Sigma_{def} = \{\texttt{diag1}, \texttt{diag2}\}$,
- $\Sigma_{open} = \{\texttt{queen}, \texttt{n}\}$,
- $\Sigma_{input} = \{\texttt{n}\}$,
- $\Sigma_{search} = \{\texttt{queen}\}$, and
- $\Sigma_{in*} = \{\texttt{diag1}, \texttt{diag2}\}$.

Thus subtask 5, **Calculating Definitions** evaluates the symbols in Σ_{in*}. The algorithm behind the **Calculating Definitions** step that sets up these individual evaluations is shown in Listing 5.3. In this listing we use the following procedures. The `calculatableinputstarsymbol` procedure returns an input$*$ symbol in the given theory for the given structure, with an added restriction on the returned symbol: any open symbol it depends on is two-valued in I_{in}. The

```
procedure calculatedefinitions(T_pre, I_in) {
  calculated = ∅;
  while(calculated ≠ Σ_in*) {
    σ = calculatableinputstarsymbol(T_pre, I_in);
    Δ = definitionof(T_pre,σ);
    I = opensymbolsof(T_pre,σ,I_in);
    σ^I = calculatedefinition(σ,Δ,I);
    I_in = addtointerpretation(I_in,σ^I);
    calculated = calculated ∪ {σ};
  }
}
```

Listing 5.3: The workflow of the **Calculating Definitions** step

`definitionof` procedure returns a definition from the given theory that contains the definition of the given symbol. The `opensymbolsof` procedure returns a limited structure that contains only interpretations necessary to evaluate the given definition. The discussion of the three above procedures is considered out of scope. As shown by the `calculatedefinition` call, for each evaluation of an input$*$ symbol σ the definition of the input$*$ symbol (Δ) as well as interpretation for which it has to be evaluated (I) is given. This call returns the calculated interpretation for σ. Afterwards, this interpretation is added to the input structure using the `addtointerpretation` procedure. The `addtointerpretation` procedure checks whether the calculated interpretation is consistent with the present pre-interpretation of σ in I_{in}. If it is consistent, the interpretation is added. If it is not consistent, the model expansion workflow is halted with the message that no model could be found. The next section provides a detailed explanation of the `calculatedefinition` procedure.

Example 5.1.2. The definitions of the diagonals in Listing 5.2 can be evaluated without performing any search, since they only depend on the $+$ and $-$ built-in arithmetic operators, and the n symbol, which is known in I_{nq}.

For the $n$queens example, there are two input$*$ symbols that have to be evaluated: `diag1` and `diag2`. The input for the evaluation of `diag1` is given in Listing 5.4. Recall that subtask 3 does definition splitting, so the definition that is given contains as few rules as possible, which causes the single definition that defines `diag1` and `diag2` in Listing 5.2 to be split up into two separate definitions that define `diag1` and `diag2` respectively.

```
structure I : Σnq {
  index = {1..4}
   diag = {1..7}
      n = {→4}
}
definition Δ =
  {
    diag1(x,y) = d ← d = x - y + n.
  }

symbol σ = diag1
```

Listing 5.4: Input for the calculation of the definition of `diag1`

5.2 Evaluating input∗ Symbols With XSB

In this section we specify how a call to `calculatedefinition`(σ,Δ,I) is resolved. We start by providing a translation of Δ and I to a Prolog program `P`. We continue by specifying which Prolog system is used and how it is configured. Finally, we provide a complete workflow for the `calculatedefinition` procedure that uses this translation to delegate this task to a Prolog system.

5.2.1 Translating FO(·) to a tabled Prolog Program

This section is concerned with translating an FO(·) definition Δ into a tabled Prolog program `P` that we can later use to evaluate that definition. More formally, this translation takes as input a definition Δ^{in} over some vocabulary Σ^{in}, as well as an input structure I^{in} over the same vocabulary. The output is a Prolog program `P` that can be queried to evaluate the definition (for the given symbol σ). Additionally, there is $\mathcal{T}^{out}$ that represents some extra constraints that the returned interpretation of σ must satisfy. More specifically, these will be the implicit function constraints in the case that σ is a function. These implicit function constraints are not translated into Prolog, but rather checked afterwards. If that check fails, then the definition does not return a valid function interpretation for the defined function σ and it is reported that no models can be found.

The input language (FO(·)) contains non-Herbrand function symbols and types. The output language (Prolog) does not. Because of this, we start with a transformation that eliminates all (non-Herbrand) function symbols and types from Σ^{in}. Next we show how to translate I^{in} and Δ^{in} into tabled Prolog clauses.

In the next sections we explain how the resulting tabled Prolog Program P can be queried to perform the evaluation of the given definition.

Eliminating Function Symbols and Encoding Types

The given definition Δ^{in} is over vocabulary Σ^{in}. Here we show how to transform the input definition (Δ^{in}) and interpretation (I^{in}) over vocabulary Σ^{in} into a more Prolog-compatible version (Δ^{out}, I^{out}, Σ^{out}) that contains only untyped predicates.

The input vocabulary Σ^{in} consists of a set of types ($\Sigma^{\mathbb{T}}$), a set of predicate symbols (Σ^{P}), and a set of (non-Herbrand) function symbols (Σ^{F}). Because Prolog is untyped we must strip all symbols of their typing before adding them to Σ^{out}. We simulate untyped predicates by working only with a single type.

We add the universal type (denoted $\mathcal{U}$) to Σ^{out} and add predicate symbols that use only this type.

For every type T in $\Sigma^{\mathbb{T}}$, we create a predicate $T_\mathtt{P}$ with associated typing $\langle \mathcal{U} \rangle$. These $T_\mathtt{P}$ are commonly known as "type predicates"; they are unary predicates with the intended interpretation that they are true only for all values in the type T. We denote the mapping of a type to its type predicate $M_{\mathbb{T}} : \Sigma^{\mathbb{T}} \to \Sigma^{T_\mathtt{P}}$, with $\Sigma^{T_\mathtt{P}}$ the set of newly generated type predicates.

For every predicate p/n in Σ^{P} we add the predicate $p_\mathtt{P}/n$ with typing $\langle \mathcal{U}, \ldots, \mathcal{U} \rangle$. We denote the mapping of a predicate to its untyped Prolog version $M_P : \Sigma^{P} \to \Sigma^{p_\mathtt{P}}$, with $\Sigma^{p_\mathtt{P}}$ the set of newly generated untyped predicates.

Because Prolog does not support non-Herbrand function symbols, we transform these to predicate symbols. For every (partial) function symbol f of arity n with an associated typing of the form $\langle T_1, \ldots, T_n, T_{out} \rangle$, we create a $n+1$-ary predicate $f_\mathtt{P}$ with associated typing $\langle \mathcal{U}, \ldots, \mathcal{U} \rangle$. We denote this mapping M_F and it has signature $M_F : \Sigma^{F} \to \Sigma^{f_\mathtt{P}}$.

Thus, Σ^{out} contains exactly **(1)** $\mathcal{U}$, **(2)** $M_{\mathbb{T}}$ (T) for all T in $\Sigma^{\mathbb{T}}$, **(3)** M_P (p) for all p in Σ^{P}, and **(4)** M_F (f) for all f in Σ^{F}.

Example 5.2.1. The vocabulary in Listing 5.4 is transformed into the vocabulary shown in Listing 5.5.

Translating the Structure

In this section, we specify how to transform I^{in} over Σ^{in} into a I^{out} over Σ^{out} that contains only predicate interpretations. We use $T_\mathtt{P}$ ($p_\mathtt{P}$, $f_\mathtt{P}$) as shorthand

```
vocabulary Σ^out_diag1 {
  type 𝒰
  index_P(𝒰)
  diag_P(𝒰)
  diag1_P(𝒰,𝒰,𝒰)
  n_P(𝒰)
}
```

Listing 5.5: Σ^{out}_{diag1} for the input in Listing 5.4

for $M_{\mathbb{T}}(T)$ ($M_P(p)$, $M_F(f)$ respectively). As a first step, we limit I^{in} to contain only the open symbols in definition Δ on which the defined symbol σ depends. We call this new temporary interpretation I^{in}_{σ}. Because of the way how `calculatableinputstarsymbol` works in Section 5.1.2, I^{in}_{σ} contains only two-valued interpretations.

The interpretation for $\mathcal{U}$ contains all domain elements that are contained in any of the types in $\Sigma^{\mathbb{T}}$. I.e., $\mathcal{U}^{I^{out}} = \bigcup_{T \in \Sigma^{\mathbb{T}}} T^{I^{in}_{\sigma}}$.

The interpretation for $T_{\mathtt{P}}$ is added to I^{out} by, for x any of the domain elements that occur in I^{in}_{σ}, setting $T_{\mathtt{P}}^{I^{out}}(x) = \mathtt{true}$ if $x \in T^{I^{in}_{\sigma}}$ and setting $T_{\mathtt{P}}^{I^{out}}(x) = \mathtt{false}$ otherwise.

The interpretation for $p_{\mathtt{P}}$ is added to I^{out} by setting $p_{\mathtt{P}}^{I^{out}}(\bar{t}) = p^{I^{in}_{\sigma}}(\bar{t})$ if tuple $\bar{t}$ respects the typing of p, and setting $p_{\mathtt{P}}^{I^{out}}(\bar{t}) = \mathtt{false}$ otherwise.

The interpretation for $f_{\mathtt{P}}$ is added to I^{out} by setting $f_{\mathtt{P}}^{I^{out}}(t_1, \ldots, t_n, t_{out}) = f^{I^{in}_{\sigma}}(t_1, \ldots, t_n, t_{out})$ if tuple $(t_1, \ldots, t_n)$ respects the input typing of f and t_{out} respects the output typing of f, and setting $f_{\mathtt{P}}^{I^{out}}(t_1, \ldots, t_n, t_{out}) = \mathtt{false}$ otherwise.

Afterwards, I^{out} is inserted into the Prolog program `P` as follows. $\mathcal{U}$ is not added to `P`. We create a mapping $\blacktriangleright_d$ that creates Prolog-compatible versions of domain elements in I^{out}. In order to support a relational operator $\sim$ as it was introduced in Section 2.2, it is essential that the domain elements produced by $\blacktriangleright_d$ reflect the Herbrand term ordering. I.e., $d_1 \sim d_2 \Leftrightarrow \blacktriangleright_d(d_1) \sim \blacktriangleright_d(d_2)$. The inverse of this mapping is later used to translate answer tuples back into IDP3 domain element tuples.

For every predicate $p \in \Sigma^{out} \setminus (\{\mathcal{U}\} \cup \{\sigma\})$ and every tuple $(d_1, \ldots, d_n)$ for which $p^{I^{out}}(d_1, \ldots, d_n) = \mathtt{true}$, add `p(`$\blacktriangleright_d(d_1)$`,...,`$\blacktriangleright_d(d_n)$`).` as a fact to `P`.

Example 5.2.2. The structure in Listing 5.4 is transformed into the output structure shown in Listing 5.6. And this output structure is inserted into `P` as given in Listing 5.7.

```
structure I^out_diag1 : Σ^out_diag1 {
  U = {1...7}
  index_P = {1...4}
  diag_P = {1...7}
  n_P = {4}
}
```

Listing 5.6: I^{out}_{diag1} for the input in Listing 5.4

```
index_P(1).
index_P(2).
index_P(3).
index_P(4).
diag_P(1).
diag_P(2).
diag_P(3).
diag_P(4).
diag_P(5).
diag_P(6).
diag_P(7).
n_P(4).
```

Listing 5.7: Prolog code for I^{out}_{diag1} in Listing 5.6

In addition to this, we have also implemented a shorter representation when it can be detected that the interpretation of a symbol is a *range* of numbers. An example of this is shown in Listing 5.8.

In the remainder of this section we discuss how to translate a given definition (Δ) over vocabulary $\Sigma^{in} = \langle \Sigma^{\mathbb{T}}, \Sigma^{P}, \Sigma^{F} \rangle$ into a Prolog program. First we perform some elementary transformations on Δ, resulting in a definition that contains rules of some basic form. Then we show how to translate each of these basic rule forms.

```
index_P(X) :- between(1,4,X).
diag_P(X) :- between(1,7,X).
n_P(4).
```

Listing 5.8: Alternative Prolog code for I^{out}_{diag1} in Listing 5.6

Transforming Δ to a basic form

Unnesting of function terms If a formula ϕ contains a nested function (or aggregate) term $f(\bar{t})$, replace ϕ with $(f(\bar{t}) = y \wedge \phi_{f \mapsto y})$ where every occurrence of $f(\bar{t})$ in ϕ is replaced with y in $\phi_{f \mapsto y}$ and y is a previously unused variable. Note that this implies unnesting applications of built-in arithmetic operators as well.

Example 5.2.3. The formula

$$p(3 + 4 - f(\#\{v : q(v)\}))$$

is transformed into

$$u = 3 + 4 \wedge x = \#\{v : q(v)\} \wedge y = f(x) \wedge z = u - y \wedge p(z).$$

After this, every function term has either a variable or a value as its argument.

This transformation is executed iteratively until all function term occurrences are part of the equality to another (variable or built-in) term $(f(\bar{t}) = t')$. In the case of multiple nestings of terms, executing this transformation from the innermost nested function term to the outermost nested function term is a way of limiting the number of additionally introduced variables.

Eliminating function symbols After the transformation above, function terms can only occur as part of an equality to another term $(f(\bar{t}) = t')$. These formulas are replaced by $f_{\mathtt{P}}(\bar{t}, t')$ with $f_{\mathtt{P}} = M_F(f)$.

If this formula was at the head of a rule, (i.e., $f(\bar{t}) = y \leftarrow \phi$.) we rewrite the rule with the new head $f_{\mathtt{P}}(\bar{t}, y) \leftarrow \phi$. In order to preserve equivalence, we must ensure that this new rule only derives interpretations for $f_{\mathtt{P}}$ that also satisfy the function constraints of f, meaning that

- there exists at most one y for every $\bar{t}$ such that $f_{\mathtt{P}}(\bar{t}, y)$ holds, and
- if f is a total function, there exists at least one y for every tuple $\bar{t}$ such that $f_{\mathtt{P}}(\bar{t}, y)$ holds.

We will call these the *function constraints* associated with $f_{\mathtt{P}}$. They are translated into FO$(\cdot)$ as shown in Figure 5.2 and are added to $\mathcal{T}^{out}$.

Eliminating equivalences All occurrences of equivalences in any body of the definition are rewritten to their implicational form.

$$\forall \bar{t} : \exists_{<2} y : f_{\mathtt{P}}(\bar{t}, y).$$
$$\forall \bar{t} : \exists y : f_{\mathtt{P}}(\bar{t}, y).$$

Figure 5.2: FO($\cdot$) expression of the function constraints on $f_{\mathtt{P}}$.

Example 5.2.4.

$$P \Leftrightarrow Q$$

is rewritten to

$$P \Rightarrow Q \wedge Q \Rightarrow P$$

Eliminating implications All occurrences of implications in any body of the definition are rewritten to their disjunctive form.

Example 5.2.5.

$$P \Rightarrow Q$$

is rewritten to

$$\neg P \vee Q$$

Pushing down negations Negations are pushed into subformulas until they are applied directly to predicate applications.

Example 5.2.6.

$$\exists x : \neg(\forall y : P(x, y) \vee Q(y, x)).$$

is rewritten to

$$\exists x : \exists y : \neg(P(x, y) \vee Q(y, x)).$$

which is further rewritten to

$$\exists x : \exists y : \neg P(x, y) \wedge \neg Q(y, x).$$

Tseitinisation [78, 85] Next, we use predicate introduction to simplify the bodies of the definitions into formulas where all subformulas are predicate applications (see Section 2.2.3). We use the following rewrite rule.

(1) For some formula $\psi[\bar{t}]$ (a formula ψ with free variables $\bar{t}$) that is not a predicate application that occurs inside a conjunction, disjunction or quantified formula of $\phi[\bar{t}]$, introduce a new predicate p_ψ of the same arity as $\psi[\bar{t}]$,

(2) substitute $p_\psi(\bar{t})$ for $\psi[\bar{t}]$ in $\phi[\bar{t}]$, and

(3) add the rule $\forall \bar{t} : p_\psi(\bar{t}) \leftarrow \psi[\bar{t}]$. to Δ.

Example 5.2.7. The rule

$$\forall x : P(x) \leftarrow (\exists y : Q(x, y)) \vee (R(x) \wedge S(x)).$$

is rewritten to the set of rules

$$\begin{array}{lcl} \forall x : P(x) & \leftarrow & p_1(x) \vee p_2(x). \\ \forall x : p_1(x) & \leftarrow & \exists y : Q(x, y). \\ \forall x : p_2(x) & \leftarrow & R(x) \wedge S(x). \end{array}$$

After these transformations, the rules in the definition have one of the following four forms, where $\bar{t}$ is shorthand for all variables in $\bar{t}_1 \ldots \bar{t}_n$:

$$\begin{array}{rcl} \textit{Conjunction} & \forall \bar{t} : p(\bar{t}) & \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n). \\ \textit{Disjunction} & \forall \bar{t} : p(\bar{t}) & \leftarrow p_1(\bar{t}_1) \vee \cdots \vee p_n(\bar{t}_n). \\ \forall \textit{ quantified} & \forall \bar{t}_1 : p(\bar{t}_1) & \leftarrow \forall \bar{t}_2 : p_\phi(\bar{t}_1, \bar{t}_2). \\ \exists \textit{ quantified} & \forall \bar{t}_1 : p(\bar{t}_1) & \leftarrow \exists \bar{t}_2 : p_\phi(\bar{t}_1, \bar{t}_2). \end{array}$$

This ***basic form*** has the following properties.

(1) Other than in a set expression, there are no function terms present and

(2) any use of negation is directly applied to an atomic formula.

We continue our translation of this basic form by showing, in order, how to translate **(1)** each of the four kinds of rules, **(2)** predicate applications in these rules, and **(3)** terms.

We indicate the translation of a term t with $\blacktriangleright_t\ (t)$, the translation of a predicate application $p(\bar{t})$ with $\blacktriangleright_p\ (p(\bar{t}))$, and the translation of a basic rule r with $\blacktriangleright_r\ (r)$.

Translating Connectives

Table 5.1 gives an overview on how to translate each of the basic forms to Prolog code.

The `bind`(S) predicate is a placeholder for calling all type predicates associated with the variables in set S. Such a `bind`$(\bar{t})$ is added at the end of every Prolog rule to ensure that all variables in the head are bound to an element of their type. If, after resolving all `bind`(S) statements in a translation, a variable is bound multiple times in a single rule, only the leftmost occurrence is kept.

Example 5.2.8. Translating the conjunctive rule

$$\forall\ x[T_1]\ y[T_2]\ z[T_3] : p(x, y, z) \leftarrow q(x) \wedge r(x, y).$$

$\blacktriangleright_r$ (*Conjunction*)	$\blacktriangleright_p\ (p(\bar{t}))$	:- $\blacktriangleright_p\ (p_1(\bar{t}_1)), \ldots, \blacktriangleright_p\ (p_n(\bar{t}_n)), \texttt{bind}(\bar{t})$.
$\blacktriangleright_r$ (*Disjunction*)	$\blacktriangleright_p\ (p(\bar{t}))$	:- $\blacktriangleright_p\ (p_1(\bar{t}_1)), \texttt{bind}(\bar{t})$.
	$\blacktriangleright_p\ (p(\bar{t}))$	:- $\vdots$
	$\blacktriangleright_p\ (p(\bar{t}))$	:- $\blacktriangleright_p\ (p_n(\bar{t}_n)), \texttt{bind}(\bar{t})$.
$\blacktriangleright_r$ ($\forall$ *quantified*)	$\blacktriangleright_p\ (p(\bar{t}_1))$	:- $\texttt{forall}(\texttt{bind}(\bar{t}_2), \blacktriangleright_p\ (p_\phi(\bar{t})))$, $\texttt{bind}(\bar{t})$.
$\blacktriangleright_r$ ($\exists$ *quantified*)	$\blacktriangleright_p\ (p(\bar{t}_1))$	:- $\blacktriangleright_p\ (p_\phi(\bar{t}))$, $\texttt{bind}(\bar{t})$.

Table 5.1: Overview of the translation of rules into Prolog code.

is transformed into

```
p(X,Y,Z) :- q(X), r(X,Y), bind({X,Y,Z}).
```

which, after resolving the `bind`(...) statement, results in the Prolog code

$$\texttt{p(X,Y,Z) :- q(X), r(X,Y), type}_{T_1}\texttt{(X), type}_{T_2}\texttt{(Y), type}_{T_3}\texttt{(Z).}$$

Where $\texttt{type}_{T_i} = M_{\mathbb{T}}(\mathrm{T}_i)$. For the code above, $\texttt{type}_{T_1}\texttt{(X)}$, $\texttt{type}_{T_2}\texttt{(Y)}$, and $\texttt{type}_{T_3}\texttt{(Z)}$ ensure that variables `X` and `Y` are type checked after they are bound by `q(X), r(X,Y)` and that all correctly typed values for variable `Z` are returned (since `Z` is not bound in the rule by anything else). For the remainder of this chapter, any Prolog code that is the result of a transformation is typeset as shown above.

The $\texttt{forall}(C1, C2)$ predicate is a predicate that only succeeds if for every succeeding call $C1$, call $C2$ succeeds as well (with identical bindings for the variables). Prolog code on how to provide this functionality can be found in Appendix A.1

As a next step, the translation of predicate applications and terms is discussed.

Translating Predicate Applications and Terms

Table 5.2 gives an overview of the translation of predicate applications and terms to Prolog. This table must be read top-down, and the first row that "matches" the syntactic structure that is to be translated must be applied.

Below each transformation is discussed in detail. We traverse the overview in Table 5.2 top-down.

$\blacktriangleright_p\ (\neg\phi[\bar{t}])$	$\rightarrowtail \texttt{bind}(\bar{t}),\ \texttt{negate}(\blacktriangleright_p\ (\phi[\bar{t}]))$
$\blacktriangleright_p\ (t_{out} = t_1 \odot t_2)$	$\rightarrowtail \bigodot_{\mathtt{P}}(\blacktriangleright_t\ (t_1), \blacktriangleright_t\ (t_2), \blacktriangleright_t\ (t_{out}))$
$\blacktriangleright_p\ (t_{out} = f(\bar{t}))$	$\rightarrowtail f_{\mathtt{P}}(\blacktriangleright_t\ (\bar{t}), \blacktriangleright_t\ (t_{out}))$
$\blacktriangleright_p\ (t_{out} = \texttt{agg}_t)$	$\rightarrowtail \texttt{setexpr(}E\texttt{,C), aggterm(agg}_f\texttt{,C,}\blacktriangleright_t\ (t_{out})\texttt{)}$
$\blacktriangleright_p\ (t_1 \sim t_2)$	$\rightarrowtail \texttt{bind}(\{t_1, t_2\}), \blacktriangleright_t\ (t_1) \sim_{\mathtt{P}} \blacktriangleright_t\ (t_2)$
$\blacktriangleright_p\ (p(\bar{t}))$	$\rightarrowtail\!= p_{\mathtt{P}}(\blacktriangleright_t\ (t_1), \ldots, \blacktriangleright_t\ (t_n))$
$\blacktriangleright_t\ (val)$	$\rightarrowtail i$
$\blacktriangleright_t\ (v)$	$\rightarrowtail \blacktriangleright_v\ (v)$

Table 5.2: Overview of the translation of predicate applications and terms into Prolog code.

$$\blacktriangleright_p\ (\neg\phi[\bar{t}]) \quad \rightarrowtail \quad \texttt{bind}(\bar{t}),\ \texttt{negate}(\blacktriangleright_p\ (\phi[\bar{t}]))$$

A **negated predicate application** is translated to a series of type predicate calls and a `\+/1` call to the translated FO(·) sub-goal. If the negated sub-goal is a defined predicate in Δ, the XSB Prolog built-in `tnot/1` is used instead. The type predicate calls are needed because `\+/1` and `tnot/1` only accept sub-goals that have no unbound variables in them.

$$\blacktriangleright_p\ (t_{out} = t_1 \odot t_2) \quad \rightarrowtail \quad \bigodot_{\mathtt{P}}(\blacktriangleright_t\ (t_1), \blacktriangleright_t\ (t_2), \blacktriangleright_t\ (t_{out}))$$

Arithmetic built-in terms are translated as part of their surrounding equality to another term. A Prolog-compatible variant of the used IDP3 arithmetic built-in is used instead. Specific code for the Prolog built-ins can be found in Appendix A.2. Here, and in some forthcoming translations, the nested terms (that are either variables or values) are translated using $\blacktriangleright_t$.

$$\blacktriangleright_p\ (t_{out} = f(\bar{t})) \quad \rightarrowtail \quad f_{\mathtt{P}}(\blacktriangleright_t\ (\bar{t}), \blacktriangleright_t\ (t_{out}))$$

An equality with an **applied function term** is translated by introducing the graph predicate associated with f ($f_{\mathtt{P}} = M_F(f)$).

$$\blacktriangleright_p\ (t_{out} = \texttt{agg}_f(E)) \quad \rightarrowtail \quad \texttt{setexpr(}E\texttt{,C), aggterm(agg}_f\texttt{,C,}\blacktriangleright_t\ (t_{out})\texttt{)}$$

An aggregate term is an aggregate function ($\texttt{agg}_f$) applied to a set expression (E). An equality with an **applied aggregate term** is translated by using a Prolog predicate that performs the requested functionality embedded in the aggregate term. This predicate takes the following approach. It first collects the list of costs (`C`) associated with the set expression E. Afterwards, the aggregate function ($\texttt{agg}_f$) is applied to `C` and a unification with $\blacktriangleright_t\ (t_{out})$ is attempted. How these operations are performed is described in detail in Appendix A.3.

$$\blacktriangleright_p\ (t_1 \sim t_2) \quad \mapsto \quad \texttt{bind}(\{t_1, t_2\}), \blacktriangleright_t\ (t_1) \sim_{\texttt{P}} \blacktriangleright_t\ (t_2)$$

Because all other syntactical cases (comparison with an aggregate term, comparison with a function term) have been filtered out at this point, the nested terms here can only be variables or values. Thus, the usage of an FO($\cdot$) **built-in relational comparison between two terms** is translated into a Prolog version of the code. The usage of `bind`/1 is needed to ensure that no comparison is done with variable terms (i.e., non-instantiated, non-ground Prolog terms).

$$\blacktriangleright_p\ (p(\bar{t})) \quad \mapsto \quad p_{\texttt{P}}(\blacktriangleright_t\ (t_1), \ldots, \blacktriangleright_t\ (t_n))$$

A **predicate application** that is not a built-in relational operator is translated using the untyped version of the predicate. Thus, $p_{\texttt{P}} = M_P(p)$.

$$\blacktriangleright_t\ (val) \quad \mapsto \quad i$$

A **value term** (val) is trivially translated. The FO($\cdot$) value terms can only be integers, these are translated into a Prolog built-in representation of that integer.

Example 5.2.9. The integer value term 5 is translated to a Prolog built-in representation of the integer 5, which is also the symbol "`5`".

$$\blacktriangleright_t\ (Variable\ v) \quad \mapsto \quad \blacktriangleright_v\ (v)$$

A **variable term** (v) is trivially translated to a Prolog variable. Every occurrence of the variable in its scope is replaced with the same Prolog variable. To ensure this, we introduce the mapping $\blacktriangleright_v$ that maps every variable occurrence in the same scope to the same Prolog variable.

Example 5.2.10. The variable term v is, in every of its occurrence, replaced with the Prolog variable `V`, where $\blacktriangleright_v\ (v) =$ `V`.

Example 5.2.11. Below is shown the translated code for the input given in Listing 5.4.

```
:- table diag1/3, diag2/3.
diag1(X,Y,D) :- n(N), type_index(X), type_index(Y),
                type_index(N), V1 is X - Y,
                V2 is V1 + N, D = V2, type_diag(D).

diag2(X,Y,D) :- type_index(X), type_index(Y),
                V1 is X + Y, V2 is V1 - 1,
                D = V2, type_diag(D).

type_diag(X)  :- between(1,7,X).
type_index(X) :- between(1,4,X).
n(4).
```

Listing 5.9: Prolog code for the diagonal definitions in Listing 5.2.

5.2.2 Choosing and Configuring a Prolog System

After providing a translation to Prolog code, we now discuss which system should be used and how some of the counterparts of the IDP3 functionality is provided in this system.

We decided to use the `XSB` Prolog system because it is the only Prolog system that, to our knowledge, implements the Well-Founded semantics as well as Tabling. The usage of the Well-Founded semantics is essential because FO(·) uses it as semantics for its definitions. The tabling aspects are essential because FO(·) supports recursive definitions.

Example 5.2.12. Consider the Prolog program given in Listing 5.10. With the query "`?- p(X).`" the execution will continually try to satisfy goal `p(b)` by calling goal `p(b)`. Tabling is needed to detect that this is a cyclical dependency and that the goal `p(b)` will never succeed.

```
p(a) :- p(b).
p(b) :- p(b).

?- p(X).
```

Listing 5.10: An example of an infinite loop in Prolog

In order to complete our translation, a few configuration steps have to be performed.

First, we have to indicate that for each defined predicate in Σ^{out}, the Prolog predicate counterpart has to be *tabled*. The tabling is not only necessary to prevent infinite execution, but also to ensure that the calculated interpretation respects the Well-Founded semantics [73]. This is done with "`:- table` $\mathtt{p_P}$`/n.`" directives where p is a defined predicate of arity n.

Additionally, it is necessary to add the flag

```
:-set_prolog_flag(unknown,fail).
```

to our Prolog code because it is possible to have no true tuples in the interpretation of open input predicates in a FO($\cdot$) model. This would result in an `XSB` program that has no fact for that predicate. Standard `XSB` detects such programs as faulty programs containing an error. Setting the flag shown above means that `XSB` returns `fail` for queries to predicate symbols that have no rules or facts.

By design of our translation of atomic formulas, negated atoms in `P` are guaranteed to be called with no unbound arguments. We replace `negate`/1 with appropriate `XSB` built-ins. If a predicate is tabled, `tnot` is used, which is filled in with negation under the WFS according to the documentation [73]. For a predicate $p_\mathtt{P}$ that is not tabled, we use negation as failure (`\+/1`). Because $p_\mathtt{P}$ will never be called recursively in any body, this also coincides with negation under the WFS.

Finally, great care must be taken in the specification of the remaining functionality that is to be provided by the Prolog system. Appendix A shows code for all necessary functionality that is added to our translated program.

5.2.3 Calling XSB from IDP3

After obtaining the resulting Prolog program `P`, the output constraints $\mathcal{T}^{out}$, and configuring the `XSB` system, we commence the interaction that returns us the interpretation for the defined predicates in Δ.

`XSB` is initiated using its `C` interface. Some necessary pre-existing built-ins are loaded using the "`?- [basics].`" directive. Next, our custom provided predicates for much of the other "built-in" functionality is loaded using the "`?- consult(builtins).`" query.

For loading `P` into `XSB`, we opted to use the file system. A `C` interface for loading clauses into `XSB` exists, but was not used mainly for ease of implementation. We continue by writing `P` to a temporary file ($\mathtt{file_P}$). However, like other Prolog

implementations, XSB provides support for both statically compiled code and dynamically asserted code. For this reason, we write the rules in P to $\texttt{file}_\texttt{P}^r$ and the facts in P to $\texttt{file}_\texttt{P}^f$. The rules in P are compiled and loaded using the "`?- consult(`$\texttt{file}_\texttt{P}^r$`).`" command. Enumerated facts are loaded dynamically using the "`?- load_dync(`$\texttt{file}_\texttt{P}^f$`).`" command. We do this because for large files containing $10^4 - 10^7$ facts, dynamic loading is much faster than XSB's compiler [73].

We then query XSB for the interpretation of the defined symbol σ/n using the `?-` $\sigma_\texttt{P}\texttt{(X}_1\texttt{,}\ldots\texttt{,X}_n\texttt{)}$ query with $\sigma_\texttt{P} = M_P(\sigma)$ if σ is a predicate, and $M_F(\sigma)$ if σ is a function symbol. Afterwards, the returned tuples can be retrieved via the C interface.

Example 5.2.13. The Prolog rules for `diag1` and `diag2` shown in Listing 5.9 are queried using the following two queries.

```
?- diag1(X,Y,D).
?- diag2(X,Y,D).
```

To process the tuples, we create a temporary interpretation of σ called $\sigma^{I'}$. The returned tuples of domain elements are translated back into their IDP3 format using the inverse of the $\blacktriangleright_d$ mapping. Thus, for every tuple $(d_1, \ldots, d_n)$ that the C interface returns, $\sigma^{I'}(\blacktriangleright_d^{-1}(d_1), \ldots, \blacktriangleright_d^{-1}(d_n)) = \texttt{true}$ is added. After all tuples have been processed, $\sigma^{I'}(d_1, \ldots, d_n) = \texttt{false}$ is added for any correctly typed tuple that was not returned by XSB. This results in a two-valued interpretation of σ. If σ was a function symbol, a check whether this interpretation satisfies all function constraints in $\mathcal{T}^{out}$ is performed. If the function constraints were not satisfied, the model expansion workflow ends and specifies that no models could be found.

As a final step, it is checked whether $\sigma^{I'} \geq_p \sigma^I$, i.e., whether the output interpretation for σ is a refinement of its input interpretation. If it was not, the model expansion workflow ends and specifies that no models could be found because the definition of σ is not consistent with the input interpretation of σ.

Example 5.2.14. Figure 5.3 shows a visual representation of the outcomes of the evaluation of the definitions of `diag1` and `diag2` respectively.

5.3 Experimental Evaluation

In this section, we compare our proposed method of adding this transformation to IDP3 (called IDP3 $^{\texttt{XSB}}$) with the current version of IDP3 and with the state-of-the-art systems CLINGO and DLV. Table 5.3 contains the results of our

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

(a) $n$queens `diag1`

4	5	6	7
3	4	5	6
2	3	4	5
1	2	3	4

(b) $n$queens `diag2`

Figure 5.3: Diagonal numbers for $n$queens puzzle with $n = 4$

experiments performed on an `Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz` with a cutoff of $500s$. Experiments that exceeded the cutoff are marked with "-". We run experiments with gringo version `4.0-rc2`, clasp version `2.1.1` and the DLV build of `Dec 17 2012`. The tools needed to run these experiments can be found on `https://dtai.cs.kuleuven.be/krr/files/experiments/IDP_XSB_experiments.tar.gz`.

Experiments were performed for REACH (reachability with undirected edges), PATH10 (calculate all pairs of nodes that are connected by a path consisting of 10 nodes), HP (Hamiltonian Path), NQ ($n$queens) and HNQ ($n$queens that computes also which placed queens are able to hit each other if they could also move like knights). All running times are given in seconds. We show for each system the grounding time (first column) and the total time (second column), except for DLV for which we only show the total time[2]. The second column in Table 5.3 indicates what portion of the predicates are input$*$ predicates. For example, i/j indicates that there are a total of j predicates and i of those are input$*$. The third column lists the problem sizes and they mean the following: for graph problems (REACH, PATH10, HP), the problem size is the number of nodes in the graph (increasing the graphs maintains edge density) and for board problems (NQ and HNQ), the problem size represents the width of the board.

For every problem except for HP, our method results in a significant decrease in running time compared to IDP3. Experiments also show that the running time complexity is of a lower order: larger problems benefit from larger speedups of

[2]DLV has a function that reports the time spent "instantiating" (i.e., grounding), but this is explicitly marked with a disclaimer that it should not be used in benchmarks

our method. For HP running times stay the same because in this problem no input$*$ predicates are used.

Note that the total time ($N = 50$: 0.79) is better than the grounding time ($N = 50$: 24.80) for the HNQ problem with the IDP3 `XSB` system. This is caused by the output predicates that are present. The grounding of output predicates (in this case, the queens that can hit each other by moving as knights) can, in the context of model expansion, be delayed until an interpretation for the remainder of the predicates (the placement of the queens) has been found. When we perform the grounding operation as a standalone operation with IDP3, these output predicates cannot be delayed and are thus also grounded, since no interpretation for the remainder of the predicates is computed. In other words, the theory splitting presented in Section 5.1 cannot be performed when performing grounding as a standalone operation. For the information in the second column, the output predicates are considered to be input$*$ predicates, since we also use `XSB` to compute their interpretation.

Experiments show that IDP3 `XSB` results in large speedups with respect to IDP3 for problems where input$*$ predicates are used. The performance of IDP3 `XSB` also lies, as can be observed, within the same order of magnitude of other ASP systems that use the semi-naive bottom-up approach, such as CLINGO and DLV. To get an idea of the communication overhead between IDP3 and `XSB`,

Problem	# input$*$	Size	IDP3		IDP3 `XSB`		CLINGO		DLV
			ground	total	ground	total	ground	total	total
REACH	2/2	10	0.06	0.06	0.09	0.09	0.01	0.01	0.01
		40	0.24	0.24	0.13	0.13	0.01	0.01	0.02
		400	106.04	106.08	1.50	1.50	0.27	0.44	4.09
PATH10	1/1	50	0.18	0.24	0.09	0.09	0.01	0.01	0.01
		300	6.00	6.00	0.25	0.25	0.01	0.01	0.04
HP	0/3	40	0.09	0.09	0.09	0.09	0.01	0.01	0.05
		400	4.15	5.00	4.15	5.00	0.03	0.15	9.34
NQ	2/3	5	0.18	0.18	0.09	0.09	0.01	0.01	0.01
		20	0.78	0.78	0.13	0.13	0.01	0.01	0.09
		100	226.07	229.09	0.39	0.58	0.10	0.20	-
HNQ	4/5	5	0.32	0.33	0.15	0.16	0.01	0.02	0.02
		20	1.52	1.53	0.80	0.18	0.02	0.03	0.23
		50	39.80	40.10	24.80	0.79	0.08	0.17	-
		100	-	-	-	4.08	0.36	0.64	-

Table 5.3: Comparison of the grounding and total execution times.

we ran programs with the transformed rules directly with `XSB` and compared it with calling them from within IDP3 `XSB`. These experiments showed that the communication between the systems causes an increase of about 30% in time spent computing the definitions.

5.4 Refining Definitions With XSB

In this section we detail how the workflow that was presented in the previous section can be slightly adjusted to offer some additional functionality. This work is a rewording of one of my published papers [55].

The previous sections explained how one can evaluate input$*$ definitions using XSB. The requirement for input$*$ symbols was there because it meant that evaluating the symbol in XSB would lead to a two-valued interpretation for that symbol. In this section, we specify how defined symbols that are **not** input$*$ symbols can be *partially evaluated* using XSB. In order to do this, we make use of the XSB mechanics that, as part of its tabling, keeps track of goals that are *undefined* because of a loop over negation during execution.

Thus, the operation discussed here is $\sigma^I = \texttt{refinedefinition}(\sigma, \Delta, I^{in})$. The outcome of this procedure is an interpretation for σ that is a refinement of the interpretation of σ in I^{in}.

The translation of the interpretation I^{in} to I^{out} is the same as described in Section 5.2.1. The difference with the `calculatedefinition` procedure that is discussed in Section 5.2 is that I^{in}_{σ} no longer has the restriction that it is a two-valued interpretation. Because I^{in}_{σ} is possibly three-valued, I^{out} may also contain some $p^{I^{out}}_{\texttt{P}}$ or $f^{I^{out}}_{\texttt{P}}$ with tuples that map to `unknown`.

The output structure I^{out} is inserted into P as follows. The universal type $\mathcal{U}$ is not added to P. We reuse the mapping $\blacktriangleright_d$ that creates Prolog-compatible versions of the domain elements used in I^{out}. We add the following rules to the start of P.

```
:- table undef/0.
undef :- tnot(undef).
```

This ensures that every rule that calls `undef` in its body ends with a tabled loop over negation. When all this is in place, I^{out} is inserted into P as follows.

For every predicate P in Σ^{out} and every tuple $(t_1, \ldots, t_n)$ for which $P^{I^{out}}(t_1, \ldots, t_n) = \texttt{true}$, add the fact "$\texttt{p(}\blacktriangleright_d\ (t_1)\texttt{,}\ldots\texttt{,}\blacktriangleright_d\ (t_n)\texttt{).}$" to P. For every predicate P in Σ^{out} and every tuple $(t_1, \ldots, t_n)$ for which $P^{I^{out}}(t_1, \ldots, t_n) = \texttt{unknown}$, add the rule "$\texttt{p(}\blacktriangleright_d\ (t_1)\texttt{,}\ldots\texttt{,}\blacktriangleright_d\ (t_n)\texttt{)}$ `:- undef.`" to P.

Example 5.4.1. Listing 5.11 shows an interpretation that is slightly adjusted from the one shown in Listing 5.2 to also include a three-valued interpretation

```
structure I_nq : Σ_nq {
      index = {1..4}
       diag = {1..7}
          n = {→4}
  queen<ct> = {}
  queen<cf> = {1,1; 2,2; 3,3; 4,4;
               1,4; 2,3; 3,2; 4,1;
              }
}
```

Listing 5.11: An example FO(·) encoding of a three-valued interpretation

```
:- table undef/0.
undef :- tnot(undef).

index_p(X) :- between(1,4,X).
diag_p(X) :- between(1,7,X).
n_p(4).
queen_p(1,2) :- undef.
queen_p(1,3) :- undef.
queen_p(2,1) :- undef.
queen_p(2,4) :- undef.
queen_p(3,1) :- undef.
queen_p(3,4) :- undef.
queen_p(4,2) :- undef.
queen_p(4,3) :- undef.
```

Listing 5.12: XSB Prolog code for the three-valued interpretation in Listing 5.11

for `queen/2` that states that it is known that no queen stands on one of the main diagonals of the grid.

Note that the XSB Prolog code only ever includes information about `queen/2` if it were ever used as open symbol in the definition of another symbol.

The translation of Δ into `P` is altered slightly. The provided `forall`(X, Y) must be changed to properly interact with the tabling aspects of `undef/0`. As before (see Appendix A.1), the `tables:not_exists`/1 XSB built-in is used for handling negation because it supports the mixed usage of tabled and non-tabled predicates (as well as the conjunction/disjunction of these). Additionaly, the call has to be surrounded by built-in "`call_tv(Call, true)`". The "`call_tv(Call, TV)`" predicate is an XSB built-in that returns only answer tuples to `Call` with the truth values `TV`. The given truth value can be `true` (`TV = true`), `false` (`TV = false`), or `unknown` (XSB knows this as `TV = undefined`). This is necessary here because otherwise, the retrieved answer may be tagged as "undefined" and interfere with our current use of the "undefined" mechanic to retrieve partial

interpretations.

This results in the following (new) Prolog code for `forall(...)`.

```
  forall(CallA, CallB) :-
    call_tv(
      tables:not_exists(
        (call(CallA),tables:not_exists(CallB))
      ),
      true
    ).
```

In addition to this, a custom `findall` predicate must be used as well when dealing with aggregates (see Appendix A.3), that takes into account the possibly `unknown` returned tuples of the nested call. This predicate works with the following strategy.

- Gather all "true" answers of the nested call.
- Gather all "undefined" answers of the nested call.
- Append each possible subset of "undefined" answers list to the "true" answers list.
- Make return tuple undefined if some "undefined" answers were added to the `Ret` list. For this, the `generate_CT_or_U_answers`/1 predicate is used. This predicate succeeds if `S` is empty and ends in `undef` if `S` is not empty.

```
ixthreeval_findall(Var,Query,Ret) :-
  findall(Var,call_tv(Query,true),CTList),
  findall(Var,call_tv(Query,undefined),UList),
  ixsubset(UList,S),
  append(S,CTList,Ret),
  generate_CT_or_U_answer(S).

generate_CT_or_U_answer([]).
generate_CT_or_U_answer([_|_]) :- undef.

ixsubset([],[]).
ixsubset([E|Tail],[E|NTail]) :-
  ixsubset(Tail,NTail).
ixsubset([_|Tail],NTail) :-
  ixsubset(Tail,NTail).
```

```
ixthreeval_findall(X,pP(X),Ret) :-
  findall(X,call_tv(pP(X),true),CTList),     % CTList = [1]
  findall(X,call_tv(pP(X),undefined),UList), % UList = [2,3]
  ixsubset(UList,S), % generates all possible subsets UList: : [],
       [2], [3], and [2,3]
  append(S,CTList,Ret), % Append the chosen subset to CTList
  generate_CT_or_U_answer(S). % If a non-empty subset was chosen,
      tag the answer as undefined
```

Listing 5.13: New code for nested `findall` calls, annotated for Example 5.4.2

Example 5.4.2. Consider the predicate $P/1$ with typing $\langle \mathbb{T} \rangle$ and an interpretation I where $\mathbb{T}^I = \{1, 2, 3\}$ and $P^I(1) = \texttt{true}, P^I(2) = \texttt{unknown}$, and $P^I(3) = \texttt{unknown}$. Consider the expression $t = \texttt{sum}\{x : P(x) : x\}$. The latter part of the equation iterates over all instances of variable x for which $P(x)$ is true, remembers the value of x, and takes the sum of all these. The expression then demands that this value is equal to t.

Because the interpretation is still three-valued, it may be refined into several (4) different two-valued interpretations. Namely,

- $\{1\}$ with $t = 1$,
- $\{1, 2\}$ with $t = 3$,
- $\{1, 3\}$ with $t = 4$, and
- $\{1, 2, 3\}$ with $t = 6$.

In order to properly refine a definition in which such an expression occurs, all values that are still possible for t must be considered. As such the Prolog code that considers this set must be adjusted as well. It must be able to return the same possible values for t. Listing 5.13 shows the new `findall` code, annotated for this example.

Thus, if the surrounding aggregate expression is able to properly use the returned `Ret` value, the following answers will be generated.

```
?- aggexpr(Out).
  Out = 1;              % S was     []
  Out = 3; (undefined) % S was   [2], used unknown values
  Out = 4; (undefined) % S was   [3], used unknown values
  Out = 6; (undefined) % S was [2,3], used unknown values
  false.
```

Which is what was needed.

Once all the above is in place, the querying for this workflow is then done as follows. In order to retrieve all the tuples in σ^I that map to `true`, the query "`?- call_tv(`$\sigma_{\mathtt{P}}$`(X`$_1$`,...,X`$_n$`),true)`" is used. In order to retrieve all the tuples in σ^I that map to `unknown`, the query "`?- call_tv(`$\sigma_{\mathtt{P}}$`(X`$_1$`,...,X`$_n$`),undefined)`" is used.

The retrieved tuples are inserted into σ^I and checked afterwards in the same manner as described in Section 5.2.3.

The behaviour of some of the `XSB` built-ins that were used is underspecified in the manual and relies heavily on how the underlying tabling mechanism is implemented. During the lifetime of the implementation of this workflow, many tweaks were made to the above code to properly support some of the corner cases. Because of this, the main goal of this implemented feature is to show that it is indeed possible to delegate this functionality to an external tabled Prolog system.

5.5 Conclusion

The grounding phase is well studied in the context of Answer Set Programming [39, 45, 74, 85]. The grounder transforms the input program into a semantically equivalent one with no variables and tries to avoid the combinatorial explosion that arises by naively instantiating the atoms in the program by all their instances. DLV and gringo (CLINGO's grounder) ground using an instantiation algorithm that is based on the well-known semi-naive bottom-up computation. They assure groundings only contain ground atoms that can be derived from the program. Moreover, ground rules are simplified by removing literals known to be true. As a consequence, the intentional predicates in safe (normal, i.e., deterministic) stratified programs are completely evaluated.

These completely evaluated predicates can be seen as our input$*$ IDP3 predicates. IDP3 is a model generator for FO$(\cdot)$ which extends first-order logic with inductive definitions and allows the programmer to write declarative specifications for his problems. For the programmer it becomes easier to give the specifications, but FO$(\cdot)$ requires additional intelligence during the grounding. The grounder of IDP3 uses a form of Lifted Unit Propagation (LUP) [80, 83] to derive extra information that can be used to reduce the grounding. LUP is effective [85], but input$*$ predicates need special treatment. In this chapter we use `XSB` and its tabling to compute the interpretations of input$*$ predicates whose first-order bodies are transformed into Prolog. Note that the safeness requirement is not needed as variables in IDP3 are typed and as such their values are known.

Additional work [39] discusses a number of optimization techniques in context of the DLV system. Some of them are relevant for our conjunctive bodies. The program rewriting strategy [38] pushes projections and selections down the execution tree, while their body reordering criterion [59] takes into account the impact on the reduction of the search space and tries to detect inconsistencies early by preferring literals with bounded variables. The techniques presented in this chapter could benefit from the use of a body reordering method. Other optimizations such as Dynamic Magic Sets [1] are related to the bottom-up strategy, while we use the tabled top-down evaluation of `XSB`.

Experiments show the proposed method results in large speedups with respect to IDP3 for problems where input$*$ predicates are used. The performance of our proposed method has the same order of magnitude of other ASP systems that use the semi-naive bottom-up approach, such as CLINGO and DLV, despite the communication overhead between IDP3 and `XSB`.

The latter part of this chapter continues this work by further exploring the coupling of IDP3 and `XSB`. In this way, we are able to deal with partial open predicates and the use of residual programs as a form of partial evaluation.

Another issue is whether, as literals in the theory get instantiated during the search phase, it is viable to evaluate these propagating definitions for these literals. This evaluation then computes the newly propagated information following from the choices the solver made. This form of goal-directed evaluation will probably require a better integration between IDP3 and `XSB` to reduce communication overhead and to benefit from the support for incremental tabling of `XSB`.

Chapter 6

Relevance for SAT(ID)

The contents of this chapters have been published in the proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16) [52] and as part of proceedings of the 9th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'16) [54].

6.1 Introduction

Since the addition of conflict-driven clause learning [66], SAT solvers have made huge leaps forward. Now that these highly-performant SAT solvers exist, research often stretches *beyond SAT* by extending the language supported by SAT with richer language constructs. Research fields such as SAT Modulo Theories (SMT) [10], CP [4] in the form of lazy clause generation [71], or ASP [62] could be interpreted as following this approach. In this chapter, we focus on the logic PC(ID): the Propositional Calculus extended with Inductive Definitions [64]. The satisfiability problem for PC(ID) encodings is called SAT(ID) [65]. SAT(ID) can be formalised as SAT modulo a theory of inductive definitions and is closely related to answer set solving. In fact, all the work we introduce in this chapter is also applicable to so-called generate-define-test answer set programs [61].

In this chapter we introduce an alternative criterion to determine satisfiability of a PC(ID) theory. Instead of searching for a variable assignment that satisfies the PC(ID) theory, we search for a *partial* assignment that contains sufficient information to guarantee satisfiability. Our approach is based on the notion

$$p_{\mathcal{T}}.$$
$$\left\{\begin{array}{ll} p_{\mathcal{T}} \leftarrow a \wedge b. \\ a \leftarrow d \vee \neg e \vee f. \\ b \leftarrow c \vee \neg g \vee h. \\ e \leftarrow f \vee \neg h \vee i. \end{array}\right\}$$

Figure 6.1: Example of a PC(ID) theory.

of *justifications* [27, 29]. As a small example, consider the theory shown in Figure 6.1.

This theory contains one constraint, that $p_{\mathcal{T}}$ must hold, and a definition (between '{' and '}') of $p_{\mathcal{T}}$ in terms of variables a to i. One way to check satisfiability would be to generate an assignment of all variables that satisfies the above theory (this is the classical approach to solving such problems). What we do, on the other hand, is to search for a *partial* assignment to these variables such that $p_{\mathcal{T}}$ is *justified* in that partial assignment. Consider for example the partial assignment where $p_{\mathcal{T}}$, a, b, c, and d are true and everything else is unknown. In this assignment, a and b are *justified* because d and c hold respectively; $p_{\mathcal{T}}$ is *justified* because both a and b are justified. This suffices to determine satisfiability of the theory, without considering the definition of e for instance.

We introduce the notion of *relevance*. Intuitively, a literal is *relevant* if it can contribute to justifying the theory. In the above example, as soon as d is assigned *true*, the variable e becomes *irrelevant*. From that point onwards, search should not take e's defining rule into account.

Based on this notion of relevance, we define two extensions of existing SAT(ID) solvers. The first is to modify the *decision heuristics*: we show that deciding on irrelevant literals *never* affects any possible justification for $p_{\mathcal{T}}$. Hence, we propose to only choose on relevant literals, otherwise leaving the heuristics unchanged. The second is to implement an *early stopping criterion* that allows a solver to decide that the theory is satisfiable from a *partial* assignment.

The contributions of this chapter are **(1)** the formal identification of the set of relevant literals, **(2)** showing that assigning a value to an irrelevant literal does not affect satisfiability, **(3)** proving correctness of the new early stopping criterion, and **(4)** experimentally evaluating the proposed approach.

The rest of this chapter is structured as follows. In Section 6.2 we present some necessary preliminaries. In Section 6.3, we present our new theory, essentially introducing relevance, the new algorithms and the associated correctness

theorems. We present a detailed description of an implementation as part of an existing SAT(ID) solver in Section 6.4. We experimentally evaluate our proposed approach in Section 6.5 and conclude in Section 6.6.

6.2 Preliminaries

6.2.1 PC(ID)

In this section, we briefly recall the syntax and semantics of Propositional Calculus extended with Inductive Definitions (PC(ID)) [63].

A truth value is one of $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$; $\mathbf{t}$ represents *true*, $\mathbf{f}$ *false*, and $\mathbf{u}$ *unknown*. The truth order $\leq_t$ on truth values is given by $\mathbf{f} \leq_t \mathbf{u} \leq_t \mathbf{t}$, the precision order $\leq_p$ is given by $\mathbf{u} \leq_p \mathbf{f}$ and $\mathbf{u} \leq_p \mathbf{t}$. Let Σ be a finite set of symbols called *atoms*. A *literal* l is an atom p or its negation $\neg p$. In the former case, we call l *positive*, in the latter, we call l *negative*. We use $\overline{\Sigma}$ to denote the set of all literals over Σ. If l is a literal, we use $|l|$ to denote the atom of l, i.e., to denote p if $l = p$ or $l = \neg p$. We use $\sim l$ to denote the literal that is the negation of l, i.e., $\sim p = \neg p$ and $\sim\neg p = p$. A *partial interpretation* $\mathcal{I}$ is a mapping from Σ to truth values. We use the notation $\{p_1^{\mathbf{t}}, \ldots, p_n^{\mathbf{t}}, q_1^{\mathbf{f}}, \ldots, q_m^{\mathbf{f}}\}$ for the partial interpretation that maps the p_i to $\mathbf{t}$, the q_i to $\mathbf{f}$, and all other atoms to $\mathbf{u}$. We call a partial interpretation *two-valued* if it does not map any atom to $\mathbf{u}$. If $\mathcal{I}$ and $\mathcal{I}'$ are partial interpretations, we say that $\mathcal{I}$ is less precise than $\mathcal{I}'$ (notation $\mathcal{I} \leq_p \mathcal{I}'$) if for all $p \in \Sigma$, $\mathcal{I}(p) \leq_p \mathcal{I}'(p)$. If φ is a propositional formula, we use $\varphi^{\mathcal{I}}$ to denote the truth value ($\mathbf{t}$, $\mathbf{f}$ or $\mathbf{u}$) of φ in $\mathcal{I}$, based on the Kleene truth tables [57]. If $\mathcal{I}$ is a partial interpretation and l a literal, we use $\mathcal{I}[l : \mathbf{t}]$ to denote the partial interpretation equal to $\mathcal{I}$, except that it interprets l as $\mathbf{t}$ (and similar for $\mathbf{f}$, $\mathbf{u}$). With σ a set of symbols, we use the notation $\mathcal{I}|_\sigma$ to indicate the *restriction* of $\mathcal{I}$ to symbols in σ. I.e., $\mathcal{I}|_\sigma(p) = \mathbf{u}$ if $p \notin \sigma$ and $\mathcal{I}|_\sigma(p) = \mathcal{I}(p)$ otherwise.

A two-valued *interpretation* I is a subset of Σ. We identify an interpretation I with the two-valued interpretation that maps $p \in I$ to $\mathbf{t}$ and $p \in \Sigma \setminus I$ to $\mathbf{f}$.

An inductive definition Δ over Σ is a finite set of rules of the form $p \leftarrow \varphi$ where $p \in \Sigma$ and φ is a propositional formula over Σ. We call p the head of the rule and φ the body of the rule. We call p *defined in* Δ if p occurs as the head of a rule in Δ with a non-empty body. The set of all symbols defined in Δ is denoted by $\mathit{defs}(\Delta)$. All other symbols are called *open in* Δ. The set of open symbols in Δ is denoted $\mathit{opens}(\Delta)$. We say that a literal l is *defined in* Δ if $|l| \in \mathit{defs}(\Delta)$. We use the *parametrised well-founded semantics* for inductive definitions [34]. That is, interpretation I is a model of Δ (denoted $I \models \Delta$) if I is the well-founded model of Δ in context $I|_{\mathit{opens}(\Delta)}$. We define the

well-founded model of a definition in context of an interpretation in the next section (Definition 6.2.2), after the concept of *justifications* has been introduced. We call an inductive definition *total* if for every interpretation I of the open symbols, the well-founded model in context I is a two-valued interpretation.

A *PC(ID)* theory $\mathcal{T}$ over Σ is a set of propositional formulas, called constraints, and inductive definitions over Σ. Interpretation I is a model of $\mathcal{T}$ if I is a model of all definitions and constraints in $\mathcal{T}$. Without loss of generality [63], we assume that every PC(ID) theory is in the **DEFNF** normal form, where $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ and

- Δ is an inductive definition defining $p_\mathcal{T}$,
- $p_\mathcal{T}$ is an atom with intended meaning that it represents the constraint that the definition defines $p_\mathcal{T}$ to be `true`,
- every rule in Δ is of the form $p \leftarrow l_1 \odot \cdots \odot l_n$, where $\odot$ is either $\wedge$ or $\vee$, p is an atom, each of the l_i are literals, and $n > 0$, and
- every atom p is defined by at most one rule of Δ.

A rule in which $\odot$ is $\wedge$, respectively $\vee$ is called a *body-conjunctive*, respectively *body-disjunctive*, rule. The rules in a definition Δ impose a *direct dependency relation*, denoted dd_Δ, between literals, defined as follows. For literals $from$ and to, it holds that $(from, to) \in dd_\Delta$ in Δ if there is a rule $p \leftarrow l_1 \odot \cdots \odot l_n$ in Δ such that for some i, either $from = p$ and $to = l_i$ or $from = \sim p$ and $to = \sim l_i$. The *dependency graph* of Δ is the graph $G_\Delta = (\overline{\Sigma}, dd_\Delta)$. For the remainder of the chapter, we assume that some PC(ID) theory $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ is fixed; hence, we will often omit Δ and/or $\mathcal{T}$ from the notations.

It has been argued many times before [26, 31, 33] that all sensible definitions in mathematical texts are total definitions. Following these arguments, in the rest of this chapter we assume Δ to be a total definition.

The satisfiability problem for PC(ID), i.e., deciding whether a PC(ID) theory has a model, is called *SAT(ID)*. This problem is NP-complete [65].

6.2.2 Justifications

Consider graph $G = (V, E)$, with V the set of nodes and E the set of edges. If the graph contains an edge from l to l' (i.e., $(l, l') \in E$), we say that l is a parent of l' in G and that l' is a child of l in G. A node l is called a *leaf* of G if it has no children in G; otherwise it is called *internal* in G. Let $G' = (V', E')$ be another graph. We define the union of two graphs (denoted $G \cup G'$) as the graph with vertices $V \cup V'$ that contains only edges that were already in G or G'.

Suppose l is a literal with $p = |l|$ and $p \in \mathit{defs}(\Delta)$ with defining rule $p \leftarrow l_1 \odot \cdots \odot l_n$. A set of literals J_d is a *direct justification* of l in Δ if one of the following holds:

- $l = p$, $\odot$ is $\wedge$, and $J_d = \{l_1, \ldots, l_n\}$,
- $l = p$, $\odot$ is $\vee$, and $J_d = \{l_i\}$ for some i,
- $l = \neg p$, $\odot$ is $\wedge$, and $J_d = \{\sim l_i\}$ for some i, and
- $l = \neg p$, $\odot$ is $\vee$, and $J_d = \{\sim l_1, \ldots, \sim l_n\}$.

Note that a direct justification of a literal can only contain children of that literal in the dependency graph.

A *justification* [29] J of a definition Δ is a subgraph of G_Δ, such that each internal node $l \in J$ is a defined literal and the set of its children is a direct justification of l in Δ. We say that J *contains* l if l occurs as a node in J. A justification is *total* if none of its leaves are defined literals. A justification can contain *cycles*. A *cycle* is a path in a graph that follows the edges in that graph and that has the same starting end ending node[1]. A cycle is called *positive* (resp. *negative*) if it contains only positive (resp. negative) literals. It is called a *mixed* cycle otherwise.

If J is a justification and $\mathcal{I}$ a (partial) interpretation, we define the value of J in $\mathcal{I}$, denoted $V_\mathcal{I}(J)$ as follows:

- $V_\mathcal{I}(J) = \mathbf{f}$ if J contains a leaf l with $l^\mathcal{I} = \mathbf{f}$ or a positive cycle (or both).
- $V_\mathcal{I}(J) = \mathbf{u}$ if $V_\mathcal{I}(J) \neq \mathbf{f}$ and J contains a leaf l with $l^\mathcal{I} = \mathbf{u}$ or a mixed cycle (or both)[2].
- $V_\mathcal{I}(J) = \mathbf{t}$ otherwise (all leaves are $\mathbf{t}$ and cycles, if any, are negative).

A literal l is *justified* (in $\mathcal{I}$, for $\mathcal{T}$) if there exists a total justification J (of Δ) that contains l such that $V_\mathcal{I}(J) = \mathbf{t}$. In this case, we say that such a J *justifies* l (in $\mathcal{I}$, for $\mathcal{T}$). We say that J *minimally justifies* l if J justifies l and there exists no subgraph J' of J that also justifies l.

Example 6.2.1. Consider justification J shown in Figure 6.2 as an example of a justification for the PC(ID) theory (in Definition Normal Form (DEFNF)) in Figure 6.1. Consider the following partial interpretations: $\mathcal{I}_1 = \{a^\mathbf{t}, c^\mathbf{t}, d^\mathbf{t}, f^\mathbf{f}\}$, $\mathcal{I}_2 = \{a^\mathbf{t}, c^\mathbf{f}, f^\mathbf{f}\}$, and $\mathcal{I}_3 = \{a^\mathbf{t}, c^\mathbf{t}, f^\mathbf{f}\}$. Literals $\{p_\mathcal{T}, a, b, d, c\}$ are *justified* in $\mathcal{I}_1$ and J *minimally justifies* $p_\mathcal{T}$ in $\mathcal{I}_1$. Justification J does not justify $p_\mathcal{T}$ for interpretations $\mathcal{I}_2$ and $\mathcal{I}_3$, because $V_{\mathcal{I}_2}(J) = \mathbf{f}$ (because $c^{\mathcal{I}_2} = \texttt{false}$) and $V_{\mathcal{I}_3}(J) = \mathbf{u}$ (because $d^{\mathcal{I}_3} = \texttt{unknown}$).

[1] In this text, we assume that Δ is finite; in this case cycles are simply "loops" in the graph. The infinite case is a bit more subtle, and an adapted definition of cycle is required to maintain all results presented below.

[2] Mixed cycles can not occur if the assumption is made that the definition is total. So-called "loops over negation" are not possible in total definitions.

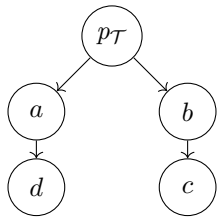


Figure 6.2: Example of a justification for the PC(ID) theory shown in Figure 6.1.

Definition 6.2.2 (Well-Founded Model)**.** An interpretation I is the well-founded model of Δ in context $I|_{opens(\Delta)}$ if for any atom p **(1)** $p^I = \mathbf{t}$ if and only if p is justified in $I|_{opens(\Delta)}$ for Δ and **(2)** $p^I = \mathbf{f}$ if and only if $\neg p$ is justified in $I|_{opens(\Delta)}$ for Δ, and **(3)** I is a two-valued interpretation.

Denecker and De Schreye [29] have shown that many semantics of logic programs can be captured by justifications. We recall their major result on the well-founded semantics.

Theorem 6.2.3 (Denecker and De Schreye [29])**.** *Let J be a justification of definition Δ.*

- *Suppose $\mathcal{I}$ and $\mathcal{I}'$ are partial interpretations. If $\mathcal{I} \leq_p \mathcal{I}'$ holds, then $V_\mathcal{I}(J) \leq_p V_{\mathcal{I}'}(J)$ also holds.*
- *Suppose $\mathcal{I}$ is an opens(Δ)-interpretation and $\mathcal{I}'$ is the well-founded model of Δ in context $\mathcal{I}$. For each defined literal l, it holds that*

$$l^{\mathcal{I}'} = \max_{\leq_t} \{V_\mathcal{I}(J) \mid J \text{ a total justification containing } l\}$$

6.3 Relevance

6.3.1 Observations

The central observation in this chapter is the fact that classical SAT(ID) solvers such as for example MiniSAT(ID) [21, 65] or the related ASP systems such as clasp [44] or DLV [60] fail to exploit an important property. Recall that a PC(ID) theory $\mathcal{T}$ of the form $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ is assumed (without loss of generalisation [22]). Systems such as MiniSAT(ID) search for an interpretation I such that $I \models \mathcal{T}$, while in fact they could search for a *partial* interpretation $\mathcal{I}$

and a *justification* J that justifies $p_\mathcal{T}$ in $\mathcal{I}$. Our claim is that even though in theory both tasks are of the same complexity, for practical applications, the latter task possesses some important advantages. Before discussing these, we provide the formal basis for our theory.

Theorem 6.3.1. *$\mathcal{T}$ is satisfiable if and only if there exists a partial interpretation $\mathcal{I}$ and a J that justifies $p_\mathcal{T}$ in $\mathcal{I}$.*

Proof. First assume that $\mathcal{T}$ is satisfiable. Then there exists an interpretation I such that ${p_\mathcal{T}}^I = \mathbf{t}$ and $I \models \Delta$. Theorem 6.2.3 (2) then yields that $\mathbf{t} = \max_{\leq_t} \{V_I(J) \mid J \text{ is a total justification that justifies } p_\mathcal{T}\}$. Hence, there must exist a justification J that contains $p_\mathcal{T}$ for which $V_I(J) = \mathbf{t}$, i.e., J justifies $p_\mathcal{T}$ in I. The result then follows by taking $\mathcal{I} = I$ and using J as justification.

On the other hand assume that there exists a partial interpretation $\mathcal{I}$ and a justification J such that J justifies $p_\mathcal{T}$ in $\mathcal{I}$. Now, let $\mathcal{I}'$ be any partial interpretation such that $\mathcal{I}' \geq_p \mathcal{I}$ and $\mathcal{I}'$ is two-valued in $opens(\Delta)$. From Theorem 6.2.3 (1) follows that $V_{\mathcal{I}'}(J) \geq_p V_\mathcal{I}(J)$, since $\mathcal{I}' \geq_p \mathcal{I}$. Because J justifies $p_\mathcal{T}$, we also know $V_\mathcal{I}(J) = \mathbf{t}$, which implies $V_{\mathcal{I}'}(J) = \mathbf{t}$. Further, $V_{\mathcal{I}'|_{opens(\Delta)}}(J) = V_{\mathcal{I}'}(J)$ since the value of a justification only depends on the edge relations in J (unchanged) and the values of open atoms (also unchanged). Let I' denote the well-founded model of Δ in context $\mathcal{I}'|_{opens(\Delta)}$. I' exists because we assume Δ to be a total definition. From Theorem 6.2.3 (2) we know that ${p_\mathcal{T}}^{I'} = \mathbf{t}$, because justification J already maps to the maximal value in the $\leq_t$ order, thus the value of the set expression in the theorem is fixed. Hence $\mathcal{T}$ is indeed satisfiable: I' is a model of $\mathcal{T}$. □

This proves that partial interpretations where $p_\mathcal{T}$ is justified are, in fact, always *partial models* [81]. We now identify which literals are *relevant*.

Definition 6.3.2 (Relevance)**.** Given a PC(ID) theory $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ and a partial interpretation $\mathcal{I}$, we inductively define the set of relevant literals, denoted $\mathcal{R}_\mathcal{T}(\mathcal{I})$, as follows

- $p_\mathcal{T} \in \mathcal{R}_\mathcal{T}(\mathcal{I})$ if $p_\mathcal{T}$ is not justified in $\mathcal{I}$,
- $l' \in \mathcal{R}_\mathcal{T}(\mathcal{I})$ if $l \in \mathcal{R}_\mathcal{T}(\mathcal{I})$, $(l, l') \in dd_\Delta$, and l' is not justified in $\mathcal{I}$.

Intuitively, a literal is relevant if making it true can help justify $p_\mathcal{T}$. If a partial structure is made more precise, literals may become irrelevant because they can no longer contribute to any justification that justifies $p_\mathcal{T}$. Often, we assume $\mathcal{T}$ is clear from the context and simply state that l *is relevant* in $\mathcal{I}$. We define the set of relevant *literals* and not the set of relevant *atoms* because, in further work, one can exploit the information that e.g., a literal l is relevant, but $\sim l$ is not.

Using relevance, we aim to obtain three advantages over classical SAT(ID) solvers.

(1) We can avoid irrelevant parts of the search space.
(2) We can stop searching once a *partial* interpretation is found in which $p_\mathcal{T}$ is justified, instead of searching for a total interpretation.
(3) We can make solvers more robust for wrong choices.

We illustrate each of these three advantages in the following example.

Example 6.3.3. Let $\mathcal{T} = \{p_\mathcal{T}, \Delta\}$ denote the theory where

$$\Delta = \left\{ \begin{array}{ll} p_\mathcal{T} & \leftarrow a \wedge b. \\ a & \leftarrow d \vee \neg e \vee f. \\ b & \leftarrow \neg h \vee j. \\ d & \leftarrow c \wedge \neg g. \\ e & \leftarrow i \vee h. \\ h & \leftarrow \neg i. \end{array} \right\}$$

Let $\mathcal{I}_1$ be the partial interpretation $\{p_\mathcal{T}{}^{\mathbf{t}}, a^{\mathbf{t}}, b^{\mathbf{t}}, c^{\mathbf{t}}, d^{\mathbf{t}}, g^{\mathbf{f}}\}$. In this case, d is justified in $\mathcal{I}_1$, hence so is a. This means that the value of e and f cannot influence whether or not a is justified. Hence, giving a value to e or to f cannot help justifying $p_\mathcal{T}$, illustrating advantage **(1)**.

Let $\mathcal{I}_2$ be $\mathcal{I}_1[j : \mathbf{t}]$. In this case $p_\mathcal{T}$ is justified in $\mathcal{I}_2$, hence Theorem 6.3.1 yields that $\mathcal{T}$ is satisfiable and we do not need to search an assignment for the remaining (irrelevant) atoms, illustrating advantage **(2)**.

Let $\mathcal{I}_3$ be $\mathcal{I}_2[e : \mathbf{f}]$. It can be seen that there exists no model of $\mathcal{T}$ that is more precise than $\mathcal{I}_3$. Indeed, e is true in every model of $\mathcal{T}$ because i as well as $\neg i$ make e true. It is possible that the solver makes the choice $e^{\mathbf{f}}$ early on. Theorem 6.3.1 shows that since $p_\mathcal{T}$ is justified in $\mathcal{I}_3$ a model must exist (even though the current interpretation is incompatible with that model), illustrating advantage **(3)**.

Example 6.3.4 (Example 6.3.3 continued)**.** The set of relevant literals for $\mathcal{I}_1$ is $\mathcal{R}_\mathcal{T}(\mathcal{I}_1) = \{p_\mathcal{T}, b, \neg h, j, i\}$. The literal $p_\mathcal{T}$ is relevant in $\mathcal{I}_1$ because it is not justified. The literal b is relevant in $\mathcal{I}_1$ since it is not justified and since $p_\mathcal{T}$, which is not justified, depends on it. The literal $\neg h$ and j are relevant in $\mathcal{I}_1$ since they are not justified and potentially useful to justify b. The literal i is relevant in $\mathcal{I}_1$ since it is not justified and might be used to justify $\neg h$. The literal $p_\mathcal{T}$ is justified in $\mathcal{I}_2$ and $\mathcal{I}_3$, which means there are no relevant literals in these partial interpretations.

Using these observations, we show how to exploit relevance to reduce the search space.

6.3.2 Exploiting Relevance

In order to exploit relevance, we assume that some search algorithm for SAT(ID) is given; we assume this algorithm searches for a two-valued interpretation I such that $I \models \mathcal{T}$. We implemented our techniques in a conflict-driven clause learning DPLL solver. However, it deserves to be stressed that all ideas developed here are independent of the choice of search strategy or heuristic. We propose the following modification to such a solver: choose only on relevant literals and stop search early if there are no unassigned relevant literals in the current search state. Note that if there are no unassigned relevant literals left, $p_{\mathcal{T}}$ is justified if and only if there is a model (according to Theorem 6.3.1). In order to prove correctness of our modification, we will use the following result.

Theorem 6.3.5. *Let $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ be a PC(ID) theory. Suppose $\mathcal{I}$ is a partial interpretation and l^{irr} a literal such that $\mathcal{I}(|l^{irr}|) = \mathbf{u}$ and l^{irr} is not relevant in $\mathcal{I}$. If $p_{\mathcal{T}}$ is justified in some partial interpretation $\mathcal{I}'$ more precise than $\mathcal{I}$, then $p_{\mathcal{T}}$ is also justified in $\mathcal{I}'[l^{irr} : \mathbf{f}]$ and in $\mathcal{I}'[l^{irr} : \mathbf{t}]$.*

Proof. Let J be a justification that minimally justifies $p_{\mathcal{T}}$ in $\mathcal{I}'$. Note that leaves in J are open and true in $\mathcal{I}'$, whereas cycles, if any, are negative.

A justification J_1 is derived from J as follows: for each defined literal x in J that is justified in $\mathcal{I}$: remove the edges from x to its children. Finally, remove all parts not reachable from $p_{\mathcal{T}}$. By construction, the leafs of J_1 are either open literals, or defined literals justified in $\mathcal{I}$.

Let J_2 be a justification that contains only literals justified in $\mathcal{I}$ and that justifies all these literals. Now, define J' as $J_1 \cup J_2$. Since J_1's internal nodes are not justified in $\mathcal{I}$, and all literals in J_2 are justified in $\mathcal{I}$, this union introduces no new loops not already in J_1 or in J_2. Additionally, J' only contains open literals already in J_1 or in J_2. This means J' is a justification that justifies $p_{\mathcal{T}}$ in $\mathcal{I}'$, since J' contains $p_{\mathcal{T}}$, leaves in J' are open and true in $\mathcal{I}'$; cycles, if any, are negative.

The justification J' cannot contain l^{irr} in the part that originated from J_1, because those are all literals that were relevant in $\mathcal{I}$. Any occurrence of l^{irr} in any part that originated from J_2 has to be an internal node, since $V_{\mathcal{I}}(J_2) = \mathbf{t}$, which demands that all leafs are true. Hence, any occurrence of l^{irr} cannot be a leaf in J', which means that changing its interpretation does not affect the value of J' in $\mathcal{I}'$. Therefore, $p_{\mathcal{T}}$ is also justified in $\mathcal{I}'[l^{irr} : \mathbf{f}]$ and $\mathcal{I}'[l^{irr} : \mathbf{t}]$. □

Theorem 6.3.5 shows that any search algorithm that can arrive in a state in which $p_{\mathcal{T}}$ is justified by deciding on a literal l that is irrelevant in its current partial interpretation, can also arrive in such a state *without* deciding on l. Hence, if a literal l is irrelevant, it is useless to choose on that literal if the goal is to justify $p_{\mathcal{T}}$. This is exactly what our proposed solver modification does: we restrict the choices of a search algorithm to the set of relevant literals.

6.4 Implementing Relevance as part of an Existing SAT(ID) Solver

This section presents the implementation of an algorithm to keep track of relevant literals.

6.4.1 The Basic Framework

As said in Theorem 6.3.1, the solver aims to arrive at a state where $p_{\mathcal{T}}$ is justified in $\mathcal{I}$. The solver does this by making decisions, performing propagation, and backtracking. To prevent the solver from making "useless" decisions, we need to know whether literals are relevant or not in $\mathcal{I}$.

We consider the underlying solver to have an internal state $\mathcal{S}$ of the form $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$, with **(1)** $\overline{\Sigma}$ denoting the set of literals used in the solver, **(2)** $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ a DEFNF theory over $\overline{\Sigma}$, and **(3)** $\mathcal{I}$ the current partial interpretation in the solver.

During the search process, the CDCL solver adds learned conflict clauses to the theory. However, learned conflict clauses are logical consequences of the theory and because of this we do not consider them to be a part of the theory $\mathcal{T}$ in $\mathcal{S}$. Instead, $\mathcal{T}$ is reserved for non-learned clauses. We assume $\mathcal{T}$ to remain static during the search process. This assumption is valid in most *ground-and-solve* systems. Recent work focuses on interleaving this process [22]. Extending this work to allow for a changing theory is future work, but should be of limited complexity given the framework we present here.

The relevance tracker needs to take into account changes in the solver state, more specifically in $\mathcal{I}$. Before we define the interface between the relevance tracker and the solver, we discuss the solver state and its changes.

During the search process, the solver iteratively performs one of the following state changes:

- $\langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle \mapsto \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[l : \mathbf{t}] \rangle$ a literal l becomes *true*, or
- $\langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle \mapsto \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[l : \mathbf{u}] \rangle$ a literal l becomes *unknown*.

Note that this set of operations allows the solver to make a literal l *false* by making literal $\sim l$ *true*.

In order to get the necessary information about the changes of the solver, the relevance tracker listens to notifications. The relevance tracker supports the following interface to the underlying solver:

- **notifyBecomesTrue(l)** a literal l becomes *true* in $\mathcal{I}$.
- **notifyBecomesUnknown(l)** a literal l becomes *unknown* in $\mathcal{I}$.
- **isRelevant(l)** query whether a given literal l is relevant (returns a Boolean value).

Methods **notifyBecomesTrue** and **notifyBecomesUnknown** must be called by the underlying solver when a literal has become true, respectively unknown. The **isRelevant** method is used by the solver to ask the tracking module whether the given literal is relevant. The relevance information allows the solver to change its underlying heuristic, selecting only relevant literals.

6.4.2 Deriving the Justification Status of Literals

The definition of relevance relies on knowledge about which literals are *justified* in the solver. In this section, we discuss how to implement the derivation of this information. We opted to implement a method that reuses the underlying SAT(ID) solver to keep track of the justification status of literals. The method creates a new atom, called the "justification atom", for each defined atom p, denoted as $j(p)$. We call a literal $j(p)$ or $\neg j(p)$ a *justification literal*.

The intended interpretation of $j(p)$ is that $j(p)$ is true if and only if p is justified, $j(p)$ is false iff $\neg p$ is justified and $j(p)$ is unknown otherwise. To ensure that justification literals indeed get the right value, an extra PC(ID) definition Δ_j, denoted the "justification definition", is added to the theory $\mathcal{T}$. The definition Δ_j is constructed based on the original definition Δ in the following manner. The existing definition Δ is copied, except that every defined atom p is replaced with the newly created atom $j(p)$. Thus, of all the atoms in the original definition, only the open atoms remain unchanged.

Example 6.4.1. Transforming the original definition

$$\Delta = \left\{ \begin{array}{rcrcrcrcr} p_{\mathcal{T}} & \leftarrow & c_1 & \wedge & c_2 & \wedge & c_3 & \wedge & c_4 \\ c_1 & \leftarrow & \neg b & \vee & \neg d & & & & \\ c_2 & \leftarrow & a & \vee & b & \vee & \neg c & & \\ c_3 & \leftarrow & \neg b & \vee & e & \vee & \neg f & & \\ c_4 & \leftarrow & d & \vee & f & \vee & \neg a & & \\ f & \leftarrow & b & \vee & d & & & & \end{array} \right\}$$

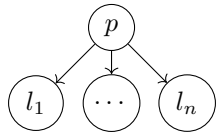
leads to the justification definition

$$\Delta_j = \left\{ \begin{array}{rcrcrcrcr} j(p_{\mathcal{T}}) & \leftarrow & j(c_1) & \wedge & j(c_2) & \wedge & j(c_3) & \wedge & j(c_4) \\ j(c_1) & \leftarrow & \neg b & \vee & \neg d & & & & \\ j(c_2) & \leftarrow & a & \vee & b & \vee & \neg c & & \\ j(c_3) & \leftarrow & \neg b & \vee & e & \vee & \neg j(f) & & \\ j(c_4) & \leftarrow & d & \vee & j(f) & \vee & \neg a & & \\ j(f) & \leftarrow & b & \vee & d & & & & \end{array} \right\}$$

In addition to the creation of this new definition Δ_j, we prohibit the solver from making choices on these justification atoms. Because of this, the value of all $j(p)$ will be purely the result of the underlying propagation mechanism for definitions. What follows is a proof that the existing propagation mechanisms will propagate *exactly* those literals that are justified. We assume a solver that performs *unit propagation* and *unfounded set propagation* [44, 64], i.e., propagation that makes all atoms in an unfounded set false.

Theorem 6.4.2. *Let Δ be a (total) definition and $\mathcal{I}$ a partial interpretation in which all defined symbols of Δ are interpreted as* **u**. *Let l be a defined literal in Δ. In this case l is justified in $\mathcal{I}$ if and only if l is derivable by **(1)** unit propagation on the completion*[3] *[17] of Δ or **(2)** unfounded set propagations for Δ in $\mathcal{I}$.*

Proof. Intuitively, from a sequence of propagations, we can create a justification and vice versa: each justification induces a sequence of propagations. The correspondence is as follows. First for the completion, if Δ contains a rule $p \leftarrow l_1 \wedge \cdots \wedge l_n$, then this rule propagates $p = \mathbf{t}$ if and only if each of the l_i has been assigned true. This corresponds to the justification



[3]The completion of a rule $p \leftarrow q$ is the underapproximation using the FO sentence $p \Leftrightarrow q$, which demands that p and q hold equal truth values

And similar justification constructs can be defined for when $\neg p$ is propagated or when the rule has a disjunctive body.

Unfounded set propagation derives that some positive loops have to be false (because they are "unfounded"), so it essentially corresponds to a justification of a set of negative facts (negations of the literals in the positive loops) by a negative cycle.

The condition that a justification can have no mixed or positive cycles corresponds to the fact that propagation must happen in order. E.g., from the rule $p \leftarrow p \vee q$, p can only be propagated if q has been assigned true; p cannot be propagated because p has been assigned true. □

The previous theorem establishes that our approach works; a justification literal $j(p)$ will be propagated to true if p is justified (note that p is justified in Δ iff $j(p)$ is justified in Δ_j). It also explains why we use a duplicated definition: the theorem only holds if $\mathcal{I}$ is an $opens(\Delta)$ interpretation. Since this cannot be enforced (we do not want to intrude in the solver's search), we make a copy and never make choices on the copied defined symbols.

Thus, we extend our solver state $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$ to a $\mathcal{S}' = \langle \overline{\Sigma}', \mathcal{T}', \mathcal{I}', \Sigma' \rangle$ with

- Σ' = set containing the newly introduced justification atoms that the solver cannot decide on,
- $\overline{\Sigma}' = \overline{\Sigma} \cup \overline{\Sigma'}$,
- $\mathcal{T}' = \{p_{\mathcal{T}}, \Delta'\}$ for $\mathcal{T} = \{p_{\mathcal{T}}, \Delta\}$ and $\Delta' = \Delta \cup \Delta_j$, and
- $\mathcal{I}'$ = a partial interpretation over $\overline{\Sigma}'$.

With all this in place, we derive the interpretation for $justified(l)$ as follows.

- $justified(p)$ is true if and only if $j(p)$ is true in $\mathcal{I}'$, and
- $justified(\neg p)$ is true if and only if $j(p)$ is false in $\mathcal{I}'$.

6.4.3 Implementing the Relevance Tracker

The source code that implements the techniques discussed here can be found online at `https://dtai.cs.kuleuven.be/krr/files/experiments/idp_relevance_experiments.tar.gz`.

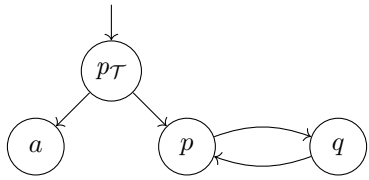


Figure 6.3: Relevance graph for $\mathcal{I} = \{\}$.

The solver maintains a subgraph of the dependency graph in order to keep track of the set of relevant literals. This subgraph, referred to as the *relevance graph*, contains all literals that are relevant and all edges between them (in the dependency graph). As such, the task of the tracker is to determine whether a given literal is a member of this graph or not. We store this graph using a data structure, denoted `candidate_parents(`l`)`, that associates a literal with a set of literals called "candidate parents". The "candidate parents" of l are the literals that are parents of l in the relevance graph. I.e., if l is irrelevant, this set is empty, otherwise, it consists of all relevant parents of l in the dependency graph. As such, it can be seen that there is an edge (p, l) in the relevance graph iff $p \in$ `candidate_parents(`l`)`. Thus, l is relevant if and only if l has a non-empty set of candidate parents. We now describe an incremental algorithm to update the set of candidate parents for all literals if the state of the solver changes. We prefer to keep these changes *local*, i.e., to not reconstruct the entire relevance graph with each solver change.

When the solver state changes and the set of candidate parents must be updated, care must be taken to detect and remove cyclic dependencies. These cyclic dependencies can arise when a candidate parent is removed from a literal l and the remaining candidate parents of that literal are not reachable from $p_\mathcal{T}$ anymore but still have l as a candidate parent, creating a loop. A more detailed example is given in Example 6.4.3

Example 6.4.3. The following definition has a cyclic dependency of the form $p \leftarrow q \leftarrow p$.

$$\left\{ \begin{array}{rl} p_\mathcal{T} & \leftarrow a \vee p \\ p & \leftarrow q \\ q & \leftarrow p \end{array} \right\}$$

Initially $\mathcal{I} = \emptyset$, thus nothing is justified and all literals are relevant. Thus, p has the set of candidate parents $\{p_\mathcal{T}, q\}$, and q has a candidate parents p.

Consider the case where a becomes true and hence $p_\mathcal{T}$ becomes justified.

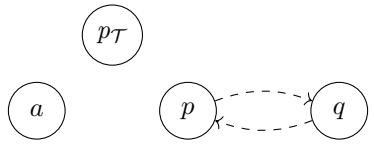


Figure 6.4: Relevance graph for $\mathcal{I} = \{a^{\mathbf{t}}\}$. Remaining loop indicated with dashed edges.

Simply removing $p_\mathcal{T}$ from the set of candidate parents of p means that p still has a candidate parent q, which is actually just a loop leading back to p. Thus, a cycle detection algorithm is needed to force p and q to remain loop-free in the relevance graph.

Thus, adding and removing candidate parents is a complicated matter. For now, we use the following interface for adjusting the set of candidate parents of a literal:

- **notifyAddCandidateParent**(l,l') to add l' to the candidate parents of l, and
- **notifyRemoveCandidateParent**(l,l') to remove l' from the candidate parents of l

The set of candidate parents for a literal potentially changes when the following changes take place (note that we already assumed the dependency relation to be immutable): **(1)** a change in the justification status of l, or **(2)** a change in the relevance status of a parent literal l'.

Thus, we extend the interface of the relevance tracker to also support the following methods.

- **notifyBecomesJustified**(l) A literal l goes from unjustified to justified
- **notifyBecomesUnjustified**(l) A literal l goes from justified to unjustified
- **notifyBecomesRelevant**(l) A literal l goes from irrelevant to relevant
- **notifyBecomesIrrelevant**(l) A literal l goes from relevant to irrelevant

In the following subsections we present **(1)** an overview of the data structures in the relevance tracker, **(2)** the algorithms for the methods in our interface, **(3)** how to optimize management of candidate parents, and **(4)** how cycle detection is done.

Data Structures

The data structures include sets and maps. Unless specified otherwise, we use hash sets and hash maps. The implementation uses `std::unordered_set` and `std::unordered_map` provided by the `C++` standard library.

Internally, we store the dependency relation dd_Δ using two maps in our module, named `children` and `parents`. These data structures map a literal to a set of literals. The first map (`children`) maps a literal to its set of children in dd_Δ. The second map (`parents`) maps a literal to its set of parents in dd_Δ. These maps are initialised using the **notifyNewRule** method. Once these maps are initialised, they remain fixed.

We use a map (`to_just_lit`) to transform a normal literal to its justification literal ($p \mapsto j(p)$, $\neg p \mapsto \neg j(p)$). For efficiency reasons, we also maintain the inverse map `to_nonjust_lit` = `to_just_lit`$^{-1}$. These maps are initialised when the justification definition Δ_j is created and do not change during execution afterwards.

We maintain a set of atoms (`is_just_atom`) to identify the justification atoms that were introduced. This set are initialised when the justification definition Δ_j is created and does not change during execution afterwards.

We use (standard) parentheses "(" and ")" to indicate the result of a map lookup, e.g.,

$$\texttt{to\_just\_lit}(p) = j(p).$$

We use (standard) parentheses to do a containment check of sets. More precisely,

$$\texttt{is\_just\_atom}(p) = \texttt{true}$$

if and only if p is in the set `is_just_atom`. As mentioned before, the underlying solver is not allowed to make decisions on literals in this set.

We maintain a map `candidate_parents` with the invariant that it maps a literal l to the set of candidate parents of l. This map is dynamic throughout execution and changes to this map are performed using the **notifyAddCandidateParent** and **notifyRemoveCandidateParent** methods.

Notification-Based Algorithms

Given that the invariant of `candidate_parents` is satisfied in solver state $\mathcal{S} = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I} \rangle$, we wish to perform the necessary changes such that they are

satisfied in solver state $\mathcal{S}' = \langle \overline{\Sigma}, \mathcal{T}, \mathcal{I}[p : tv] \rangle$ with p some atom and tv one of the truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

We initiate our notification-based algorithm as follows. If $tv = \mathbf{t}$, then we call **notifyBecomesTrue**(p). If $tv = \mathbf{f}$, then we call **notifyBecomesTrue**($\neg p$). If $tv = \mathbf{u}$, then we call **notifyBecomesUnknown**(p).

This call, in turn, can trigger other internal notifications. The implementation of these cascading notifications ensures that `candidate_parents` will comply with its invariant in interpretation $\mathcal{S}'$ after the designated call to **notifyBecomesTrue** or **notifyBecomesUnknown** is complete.

The relevance tracker implements **isRelevant**(l) by checking whether `candidate_parents(`l`)` maps to an empty set or not. This is a correct representation of the relevance status of l if the invariant of `candidate_parents` is satisfied.

For methods **notifyBecomesTrue**(l), **notifyBecomesUnknown**(l): the given literal can be a normal literal (p or $\neg p$) or a justification literal ($j(p)$ or $\neg j(p)$). The relevance tracker takes no action for normal literals. If the given literal is a justification literal, then we retrieve the original normal literal and notify the relevance tracker that this literal has become (un)justified. Note that we reuse the notation of $|l|$ to indicate the atom of literal l. In the below explanation the helper methods **notifyRemoveAllCandidateParentsOf** and **notifyAddCandidateParent**(l,p) are used. These methods are further discussed in the next section.

notifyBecomesTrue(l): if `is_just_atom(`$|l|$`)`, then call **notifyBecomesJustified**(`to_nonjust_lit(`l`)`).

notifyBecomesUnknown(l): if `is_just_atom(`$|l|$`)`, then call **notifyBecomesUnjustified**(`to_nonjust_lit(`l`)`).

notifyBecomesJustified(l): redirect to **notifyRemoveAllCandidateParentsOf**(l).

notifyBecomesUnjustified(l): for all parents p of l that are relevant, call **notifyAddCandidateParent**(l,p).

notifyBecomesRelevant(l): for all children c of l, call **notifyAddCandidateParent**(c,l).

notifyBecomesIrrelevant(l): for all children c of l, call **notifyRemoveCandidateParent**(c,l).

Maintaining Watches Instead of Sets of Candidates

The above methods dictate how the `candidate_parents` map should be manipulated. For efficiency reasons, the relevance tracker does not actively maintain this set of candidate parents. Instead it keeps track of a single candidate parent as "watched" parent. This watched parent is maintained using a map called `watched_parent(`l`)` that maps a literal to a single parent of l. The method **isRelevant(**l**)** now checks whether a given literal l has a watched parent or not.

We only keep track of a single watched parent in order to minimize how many times a cycle detection algorithm has to be invoked. The manipulation of the set of candidate parents, along with the invocation of a cycle detection algorithm is done as follows

- **notifyAddCandidateParent(**l,l'**)** Check for the following criteria:
  - l does not have a watched parent yet,
  - l is not justified,
  - l' is relevant, and
  - l is a child of l'.

  If they are met, make `watched_parent(`l`) =` l' and call **notifyBecomesRelevant(**l**)**. Note that a cyclic dependency check between l and l' is not needed, since l could not have been a suitable watch for any other literal, as it was not relevant before.

notifyRemoveCandidateParent(l,l') If l had l' as its watch, remove l' as watched parent of l. Try to find an alternative candidate parent n such that the following hold:

- $n \neq l'$,
- l is a child of n,
- l is not justified,
- n is relevant, and
- use a cycle detection algorithm to verify that `watched_parent(`l`) =` n would not create a cyclic dependency.

If such n can be found, set `watched_parent(`l`) =` n. If such n cannot be found, call **notifyBecomesIrrelevant(**l**)**.

The implementation of maintaining a suitable watched parent is a reuse of the existing "unfounded set detection" algorithm. Modern ASP solvers also make use of *source pointers* [42]) as watches for detecting unfounded sets. As such, the *relevance graph* is similar to the *source pointer configuration*, since the latter also imposes acyclicity constraint. This algorithm is considered to be the fastest algorithm to achieve this task to date.

Detecting Cycles

For our implementation of the detection of cycles, we reuse parts of the existing *unfounded set propagation* algorithm [44, 64]. This algorithm has a subcomponent that searches for cycles over negative literals.

6.5 Experimental Evaluation

In order to empirically evaluate our proposed approach, we adjusted the IDP3 system [19] and its underlying solver MiniSAT(ID) [21] to take relevance into account as described in Section 6.4. Integrating relevance into the search process is simple: it is a non-intrusive modification to the search heuristic to not choose on certain literals. However, calculating which literals are relevant requires a tight integration with the solver being adapted. Detailed information about the solver state, such as the dependency graph and the justification status for literals, are required in order to calculate which literals are relevant. For the purpose of this chapter, we opted for a simple and non-intrusive implementation that had the drawback of significant overhead. Therefore, search space size rather than solving time is measured. This performance overhead is not inherent to maintaining relevance. Large parts of the bookkeeping we do now is discovering information that is already present somewhere in the solver internally. However, extracting all the necessary information is an engineering task we did not complete yet. In this section, we will answer the following questions to evaluate whether it is worth investigating relevance further:

(Q1) How often does the VSIDS, the current state-of-the-art heuristic for SAT, make irrelevant decisions?

(Q2) Can we improve the performance of SAT(ID) solvers using relevance?

The complete set of experiments and information on how to run them can be found at `https://dtai.cs.kuleuven.be/krr/files/experiments/idp_relevance_experiments.tar.gz` .

Problem	#	μ_{irr^d}	$\sigma^2_{irr^d}$	μ_{irr^c}	$\sigma^2_{irr^c}$
GG	0/30	-	-	-	-
HP	102/102	27.37%	2.87%	36.99%	7.88%
*n*queens	14/29	22.55%	0.11%	0.43%	0.00%
PPM	13/30	22.93%	5.10%	4.98%	0.00%
RR	0/30	-	-	-	-
Sokoban	4/30	48.20%	7.62%	0.96%	0.01%
Solitaire	17/27	13.32%	0.13%	3.95%	0.19%
SM	27/30	96.40%	0.13%	0.01%	0.00%
Visit All	19/30	15.02%	2.16%	16.45%	3.42%

Table 6.1: Statistics per problem: the columns represent number of instances solved, percentage of irrelevant decisions (mean μ and variance σ^2), and percentage of irrelevant decisions in conflicts (mean μ and variance σ^2). GG = Graceful Graphs, HP = Hamiltonian Path, PPM = Permutation Pattern Matching, RR = Ricochet Robots, SM = Stable Marriage.

For these experiments we selected problems from previous ASP competitions that could be encoded without the use of aggregates and functions, since we do not yet support these language constructs. We ran these problems on an **Intel(R) Xeon(R) CPU E5645 @ 2.40GHz** CPU, using a time limit of 7200 seconds and a memory limit of 8GB.

To answer **(Q1)** we ran all the above problems and their instances with a solver configuration that uses the VSIDS heuristic while keeping track of relevance. We keep track of whether the decision made by VSIDS is relevant without actually preventing decisions on irrelevant literals (i.e., the search behaviour is **not** affected). Table 6.1 shows the problems and the numbers of successfully solved instances versus total number of instances (second column). In this table we show the mean (μ) and variance (σ^2) of **(1)** irr^d: the ratio between the irrelevant decisions made by VSIDS and the total number of decisions, and **(2)** irr^c: the ratio between the number of irrelevant decisions involved in conflicts and total number of decisions involved in conflicts. In order to obtain the latter statistic, we analyse the conflicts that occur during solving by applying full resolution on them. The resulting clause only contains decision literals. We then count the total number of decisions as well as the number of decision literals that were irrelevant at the time they were made.

Due to our significant performance overhead in keeping track of relevance, we were not able to solve a single instance of the Graceful Graphs and the Ricochet Robots problems. We observe that the VSIDS heuristics chooses a considerate amount of irrelevant literals, on average. There is even an outlier in the Stable Marriage problem where more than 96% of the choices were irrelevant. Therefore we can say for **(Q1)** that VSIDS selects, on average, a significant amount of irrelevant choice literals on the classical benchmarks.

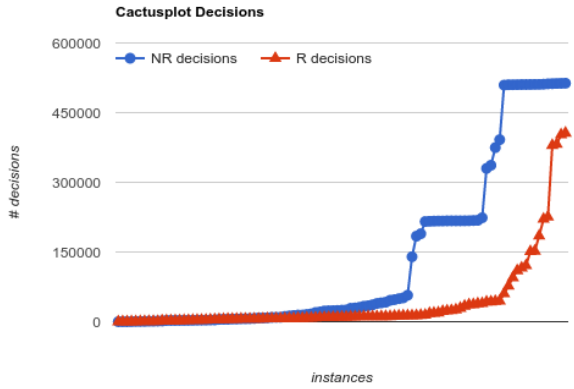


Figure 6.5: Cactusplot of # decisions.

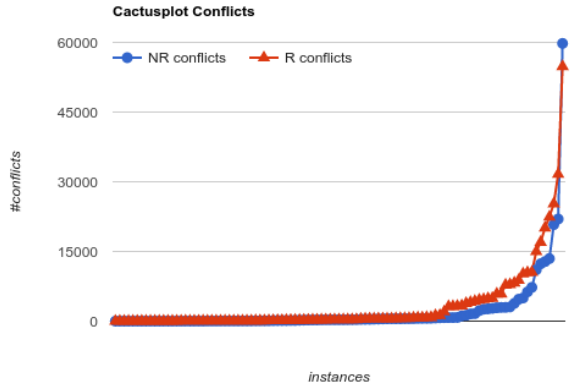


Figure 6.6: Cactusplot of # conflicts.

On the other hand, irr^c is generally significantly lower than irr^d, meaning that the irrelevant decisions made by VSIDS hardly ever lead to conflicts. In order to further inspect the behaviour of relevance we discuss cactusplots for the behaviour of the experimental runs in Table 6.1 for instances that were solved both by VSIDS (labeled "NR", for "No Relevance") and by our proposed solver modification (labeled "R", for "Relevance").

Figure 6.5 shows that we succeeded in reducing the number of decisions made, and Figure 6.6 shows that this did not affect the number of conflicts for these benchmarks. This initial observation is not encouraging, since the number of

conflicts is often taken as a measure for the size of the search space traversed. In what follows, we

(1) argue that in certain applications, reducing the number of decisions is already a desirable property as such, and
(2) investigate why we observe no reduction in the number of conflicts.

Reducing decisions: a contribution on its own Even if we did not manage to significantly reduce the number of conflicts, reducing the number of decisions is already a significant achievement for certain applications. To illustrate this, we consider lazy model expansion [22]. The approach of lazy model expansion is to interleave the grounding and the search phases. That is, a first-order theory is not translated to propositional logic a priori. Instead, depending on the search of a SAT(ID) solver, certain parts of the grounding are generated. This approach works (roughly) as follows. A PC(ID) theory $\mathcal{T}$ is initialised as $p_{\mathcal{T}}$. Each time a literal that has no definition in $\mathcal{T}$ is assigned a value, some external procedure is called and the definition of that literal is added to $\mathcal{T}$. This approach is particularly fruitful in applications with very large (possibly infinite) domains where it is simply infeasible to generate the entire grounding.

Adding more definitions to $\mathcal{T}$ is possibly a costly operation and should be avoided as much as possible. If we combine lazy grounding with our proposed relevance approach we will greatly benefit from the reduced number of decisions made, because avoiding irrelevant decisions results in fewer variables that are assigned a value (also propagations that follow from irrelevant decisions!) and hence less grounding.

Analysing the conflict behaviour We noticed that, while VSIDS makes lots of choices on irrelevant literals, the number of conflicts did not increase significantly. One possible explanation for this behaviour is that in the examples we used, the irrelevant parts of the search space are not strongly constrained. One real-world example of a problem where irrelevant parts of the search space are still heavily constrained is a scheduling problem for a trucking company, such that each scheduled truck can solve a packing problem. Solutions to such problems are often hand-made in such a way that they take relevance into account (i.e., first solving the scheduling and then only trying to solve the relevant packing problems), because the current generation of solvers cannot handle this problem directly. An instance of such a problem and its solution is given by Verstichel [82]. In order to test the hypothesis that underconstrained problems are indeed at the root of this behaviour, we construct a small encoding in which we force irrelevant literals to represent that a combinatorially hard probem is satisfiable:

$$\forall x[1..n] : XOR(x) \Leftrightarrow (P(x) \Leftrightarrow \neg Q(x)).$$
$$\forall x[1..n] : XOR(x) \Rightarrow pigeon_{k,k}.$$
$$\forall x[1..n] : \neg XOR(x) \Rightarrow pigeon_{k,k+1}.$$

Figure 6.7: Hand-made encoding showing the use of relevance. For ease of reading, a first-order version of the encoding is presented.

Figure 6.7 presents an encoding of the following problem. Predicates P and Q can be chosen freely (they are open variables of the underlying definition). For each domain element d, $XOR(d)$ holds if and only if exactly one of $P(d)$ and $Q(d)$ holds. Next, if $XOR(d)$ is $\mathbf{f}$, an encoding of an unsatisfiable pigeonhole formula must be satisfied. If $XOR(d)$ is $\mathbf{t}$, an encoding of a satisfiable pigeonhole formula must be satisfied. Thus, the problem can only be solved by making $XOR(d)$ $\mathbf{t}$ for all instances. At any point during search, VSIDS can make choices on the variables occurring in the encoding of the pigeonhole problems. As soon as XOR is decided, the relevance heuristic, on the other hand, only makes choices on variables in the relevant subproblem. If unlucky, VSIDS behaviour can lead to a great deal of time wasted and a great number of unnecessary conflicts during search.

In order to test the behaviour of VSIDS on this problem we used the same setup as in Table 6.1. This time we also measured the solving time and memory needed, as well as the total number of decisions and conflicts. We ran the above encoding with the domain of size $n = 250$ and $k = 9$. The results presented in Table 6.2 show that there are problems where taking relevance into account leads to a greatly reduced number of conflicts, which indicates a reduction of the search space. Increasing the domain size only widened the gap between VSIDS and relevance. Runtime statistics of such additional experiments are omitted here, for brevity concerns. These observations lead to a definite positive answer to **(Q2)**.

	VSIDS	**Relevance**
Running time (ms)	35691	**12523**
Memory (MB)	**192**	217.1
# decisions	10317218	**150851**
# conflicts	116434	**20900**
% irr^d	96.15%	0.00%
% irr^c	96.49%	0.00%

Table 6.2: Performance of VSIDS vs. Relevance on the hand-made problem encoding shown in Figure 6.7.

6.6 Conclusion

In this chapter we formally identified a set of literals being *relevant*; we showed that irrelevant literals cannot influence the justification status of a PC(ID) theory and hence making choices on irrelevant literals is useless with respect to proving the satisfiability of the given PC(ID) theory. We proposed two simple solver modifications: choosing only on relevant literals and stopping early. A detailed description on how to implement these modifications in an existing SAT(ID) solver is given.

In this chapter we provided a preliminary experimental evaluation using a simple and non-intrusive implementation of these proposed modifications. We compared our algorithms with the VSIDS heuristics, the current state-of-the-art heuristic for SAT solvers.

Our conclusions are that, in the benchmarks that we ran, VSIDS was observed to choose on a significant amount of irrelevant literals. As such, our proposed solver modification to VSIDS successfully managed to decrease the number of decisions made. However, we were not able to significantly reduce the number of conflicts, which would mean a reduction in the search space. Our hypothesis as to why the number of conflicts did not decrease with the number of decisions was confirmed using a crafted example. Furthermore, we sketched situations in which the decrease in the number of decisions alone is significant enough to improve performance compared to the current state-of-the-art.

The notion of relevance is related to Magic Sets [9, 11] in the field of Logic Programming in the sense that the resulting program of the magic set transformation on a program P and query Q does not execute parts of P that do not contribute towards solving query Q. In a similar fasion, the detection of SCC's in most modern SAT and ASP solvers also aims to achieve a similar goal: detect parts of the problem that do not need to be processed in order to provide a solution.

ASP grounders [39, 45] often employ such an SCC analysis on the non-ground input program that may even determine that some parts of the input specification do not even need to be ground. This is also achieved by the extended workflow presented in Section 5.1.1, where the **Forget** part of a theory is not processed at all. This can be seen as a "lifted" form of relevance, because these techniques derive a higher level that certain parts will always be *irrelevant* at the ground level.

One area where relevance should give great speedups is stable model counting. When $p_{\mathcal{T}}$ is justified, the number of solutions that this partial assignment represents is equal to 2^n with n the number of unassigned open atoms in Δ.

Stable model counters generally stop when the *justified residual program* [6] is empty. Whenever this occurs, $p_{\mathcal{T}}$ is justified. However, this does not hold the other way round. There are other cases where $p_{\mathcal{T}}$ is justified, but the justified residual program is non-empty. As such, exploiting relevance can ensure cutting out bigger parts of the search tree when counting models.

Our notion of irrelevance is closely related to "don't care atoms" in satisfiability solving [40]. However, there is an important difference between don't cares and irrelevant literals. To complete a partial structure with don't cares, *any* value may be assigned to a don't care literal; to an irrelevant literal, on the other hand, we only know that *some* value can be found for it. The value for irrelevant literals can be found as follows: first, *any* value can be assigned to the irrelevant atoms that are open in Δ. Given these values to the opens, the (parametrised) well-founded model of Δ can be computed in polynomial time. The value of any other irrelevant literal is its value in the well-founded model. This is exactly what happens in the proof of Theorem 6.3.1.

We believe that further research into relevance will be of great value and see several topics for future work. First of all, the current theory is limited to PC(ID): further language extensions such as aggregates and arithmetic are not yet supported. Second, our theory also applies to generate-define-test ASP programs; experimentally evaluating relevance in a native ASP solver can yield interesting results. Third, engineering a more efficient algorithm to keep track of relevant literals can shed light on the possible (time-wise) performance gains resulting from relevance. Fourth, experimentally evaluating relevance in the context of lazy grounding is needed to verify our hypothesis that relevance can yield great improvements there.

Chapter 7

Conclusion

In this chapter we summarize the main contributions and conclusions of this text. After this, some directions for future work are given.

7.1 Contributions and Conclusions

The main goal of this text was to better understand and provide improvements for the IDP3 system, the only known implementation of the Knowledge Base System (KBS) paradigm. IDP3 is a state-of-the-art declarative solving system that is based on a *grounding* phase and a (SAT) *solving* phase. This text contains an evaluation of existing grounding and solving techniques, as well as several additions to either the grounding or the solving part of IDP3.

- The chapters containing the preliminaries (Chapters 2 and 3) served as an introduction for understanding and developing a KBS. These chapters introduced concepts going from core mathematical constructs (such as sets, types ...) to the current state of the IDP3 KBS.
- The preliminaries gave a high-level overview on several existing grounding techniques for FO($\cdot$) specifications (Section 3.2.1). These techniques were later extensively evaluated for their benefits (see Section 4.1), which lead to a discussion on which grounding techniques are most difficult to implement versus which techniques offer the most advantages with respect to performance during grounding.

- Grounding differently leads to several representations that encode the same problem. Sometimes, unnecessary elements are present in these representations, increasing them in size. There are grounding techniques to prevent unnecessary elements from being present, thus ensuring a smaller representation. Chapter 4 contains an experiment that determines the effect of the use of such techniques on the performance of the solver that is called afterwards with this grounding (see Section 4.2). We concluded that the size of the grounding has impact on the overhead during solving. However, a rigorous statistical analysis also showed that the size does **not** have a significant impact on the search tree. This creates a clear picture for grounders that translate to input of a CDCL-based solver. As long as the grounding fits in the memory and can be created fast enough, the solver is still able to properly use the CDCL mechanism and the VSIDS heuristic to speed up search.

- In Chapter 5 we discussed a more complex version of the workflow of IDP3. We identified a sub-step in this workflow that relies on evaluating definitions efficiently. A translation of the input of this definition evaluation step into a Logic Programming (LP) problem was described. We then showed how, using the `XSB` tabled Prolog system, this translation can be used to perform definition evaluation. Experiments showed that this addition puts us on even foot compared to other state-of-the-art ground-and-solve systems for problems that rely on definition evaluation. As an extension to this, we also showed how to *refine* a more general class of definitions.

- An extension to the solver of IDP3 was discussed in Chapter 6. In this chapter, we identified a formal property (*relevance*) of literals in the solving process. Next, we proved that literals that have this property cannot possibly contribute to the search process for solutions that would originate from the current partial assignment in the solver. A detailed description of an implementation that exploits this property that extends an existing SAT(ID) solver is also given. Experiments showed that exploiting this property enables the solver to prevent a significant amount of decisions. The experiments also showed that the current implementation, however, still suffers from a substantial overhead cost.

Given all these findings, this work has provided a further refined version of an implementation of the Knowledge Base System (KBS) paradigm. The IDP3 system had already been used as an application by third-party end-users [16]. However, some other applications were only possible because of the material presented in this thesis [24, 25]. Although there are still opportunities to improve the system, it is my belief that IDP3, in its current state, has never been more

ready to be confronted with challenges originating from industrial applications. We hope that, in this way, the work in this text further strengthens the position of declarative programming, and especially the KBS paradigm, as an alternative approach to software development.

7.2 Future Research Ideas

Industrial-scale applications often require some practical feature that, whilst considered trivial from a theoretical setting, may have a significant implementation cost. Examples of such features are **(1)** support for floating point numbers and arithmetic operations on them, **(2)** the capability for interfacing with other, external, systems, and **(3)** the integration of existing databases as containers of knowledge. Implementation of these features is essential towards strengthening the position of IDP3 as a practical alternative to existing software development techniques.

Another area of future research challenges lies in the further exploration of the techniques presented in this thesis.

- We have observed in Chapter 6 that VSIDS chooses on a significant amount of irrelevant literals. However, the number of *conflicts* that are encountered in the solving process could not be reliably reduced by exploiting the relevance property. This observation warrants further investigation. Why does VSIDS *still* perform as well as it does compared to our proposed alternative, even when we have observed that it does a significant amount of "useless" work?

- A category of problems for which *relevance* detection would provide us with a significant performance advantage was presented in Section 6.5. An interesting next step would be to set up new experiments to examine the effect of exploiting relevance on this category of problems.

- In a same line of reasoning, we hypothesize that our *relevance* technique would perform well when combined with the *lazy grounding* technique. Further research that implements support for this combination and experimentally evaluates it is considered a logical next step.

- The current implementation for *relevance* is an effort into which only several man-months were invested. The current generation of state-of-the-art SAT solvers has had more than a decade's worth of engineering effort put into it. If we wish to compete with these technologies and further argue that this technique is essential to a newer generation of solvers,

additional engineering effort into a more efficient implementation of this technique is required.

Appendix A

Additional XSB Prolog code

In this appendix we list Prolog code for some of the functionality we rely on in our translation of FO(·) to Prolog (Section 5.2.1). This appendix is further divided in sections that each address one aspect of functionality that was necessary in the translation. The reader is assumed to be able to read Prolog code.

A.1 The `forall/2` Predicate in XSB

The `forall`() predicate is provided as follows.

```
forall(CallA,CallB) :-
  tables:not_exists((call(CallA), tables:not_exists(CallB))).
```

The `tables:not_exists/1` XSB built-in is used for handling negation because it supports the mixed usage of tabled and non-tabled predicates and queries (as well as the conjunction/disjunction of these).

A.2 Translation of IDP3 Built-in Arithmetic Operators to XSB Prolog Code

This appendix is an addition to Section 5.2.1 that discusses how built-in arithmetic operators ($\blacktriangleright_p$ $(t_{out} = t_1 \bigodot t_2)$) in IDP3 are translated into XSB Prolog. The trivial implementation is to call the type

predicates of t_1 and t_2 to ensure that they are bound. Afterwards, the numerical operation can be executed using the built-in predicates for $+$ (`prolog_plus()`), $-$ (`prolog_min()`), $\times$ (`prolog_times()`), / (`prolog_div()`), and % (`prolog_mod()`), which are defined as follows.

```
prolog_plus(Tout,T1,T2) :- Tout is T1 + T2.
prolog_min(Tout,T1,T2) :- Tout is T1 - T2.
prolog_times(Tout,T1,T2) :- Tout is T1 * T2.
prolog_div(Tout,T1,T2) :- Tout is T1 / T2.
prolog_mod(Tout,T1,T2) :- Tout is T1 mod T2.
```

Some intelligence can be provided in these predicates. Below is an example of a more intelligent version of `prolog_div()`.

```
% prolog_div(Solution,Numerator,Denominator)
% Represents the built-in "Solution is Numerator/Denominator"
% Handle special cases first: X is Y/X, Y known
prolog_div(Denominator,Numerator,Denominator) :-
  nonvar(Numerator),
  var(Denominator),
  Denominator is sqrt(Numerator).

% Handle special cases first: X is Y/X, X known
prolog_div(Denominator,Numerator,Denominator) :-
  var(Numerator),
  nonvar(Denominator),
  ixdifferent_number(Denominator,0),
  Numerator is Denominator*Denominator.

% Handle special cases first: X is X/Y, Y known
% -> infinite generator
prolog_div(Numerator,Numerator,1) :-
  var(Numerator),
    throw_infinite_type_generation_error.

% Handle special cases first: X is X/Y, Y known, with Y ~= 1
% -> this always fails
prolog_div(Numerator,Numerator,Denominator) :-
  nonvar(Denominator),
  ixdifferent_number(Denominator,1),
  fail.

% Handle special cases first: 0 is 0/Y with Y known
% -> succeeds
prolog_div(Numerator,Numerator,Denominator) :-
  nonvar(Numerator),
  ixsame_number(Numerator,0),
  nonvar(Denominator),
  ixdifferent_number(Denominator,0).

% Only normal cases left: X is Y/Z with Y and Z variables
prolog_div(Solution,Numerator,Denominator) :-
  nonvar(Numerator),
```

```
  nonvar(Denominator),
  TMP is Numerator / Denominator,
  ixsame_number(TMP,Solution).
```

A.3 Translation of IDP3 Aggregate Terms to XSB Prolog Code

This appendix is an addition to Section 5.2.1 that discusses how aggregate terms ($\blacktriangleright_p$ ($t_{out} = \mathtt{agg}_t$)) in IDP3 are translated into XSB Prolog. An aggregate term with free variables $\bar{y}$ is of the form

$$\mathtt{agg}_f\{\bar{x} : \phi[\bar{x},\bar{y}] : t[\bar{x},\bar{y}]\}$$

with $\mathtt{agg}_f$ one of # (cardinality), max (maximum element), min (minimum element), sum (sum of all elements), prod (product of all elements). First, it is rewritten to unnest the cost term into

$$\mathtt{agg}_f\{\bar{x} : (\phi[\bar{x},\bar{y}] \ \wedge \ z = t[\bar{x},\bar{y}]) : z\}$$

with z a new and unique variable. Next, one collects the list of costs (`C`) associated with the set expression using the following code.

```
aggexpr_P(ȳ,Out) :-
  findall((C,x̄), (
    φ_P(x̄,ȳ,z),
    z = C)
  ), Tuples),
  extract(Tuples,Costs),
  agg_f(Costs,Out).
```

The `findall/3` predicate is a built-in predicate of `XSB` that provides the functionality that we need here. The variables $\bar{y}$ are the free variables that are bound prior to accessing this code. The $\phi_{\mathtt{P}}(\bar{x},\bar{y})$ call iterates over all bindings of $\bar{x}$ that, given the bindings for $\bar{y}$, satisfy formula ϕ. The unification z `= C` binds `C` to the variable z that was introduced earlier to represent the cost. Note that the first argument of `findall/3` has been constructed in such a way that `Tuples` has the costs as its first argument. Next, `extract(Tuples,Costs)` is a Prolog built-in that transforms a list of the form `(C`$_i$`,[1,...,n])` to a list (`Costs`) containing only the costs `C`$_i$.

```
extract(Tuples,Costs) :-
  findall(C,member((C,_),Tuples),Costs).
```

Afterwards, the aggregate function ($\mathtt{agg}_f$) is applied to `C` and a unification with $\blacktriangleright_t$ (t_{out}) is attempted. These operations are executed using `XSB` as follows. In this code, the following `XSB` counterparts for aggregate functions are used:

- # (cardinality): `aggcard`,
- max (maximum element): `aggmax`,
- min (minimum element): `aggmin`,
- sum (sum of all elements): `aggsum`, and
- prod (product of all elements): `aggprod`.

```
aggcard(List,Card) :- length(List,Card).

aggmax([X|Rest],Max) :- aggmax(Rest,Max,X).
aggmax([],Max,Max).
aggmax([X|Rest],Max,TmpMax) :-
  X > TmpMax,
  aggmax(Rest,Max,X).
aggmax([X|Rest],Max,TmpMax) :-
  X =< TmpMax,
  aggmax(Rest,Max,TmpMax).

aggmin([X|Rest],Min) :- aggmin(Rest,Min,X).
aggmin([],Min,Min).
aggmin([X|Rest],Min,TmpMin) :-
  X < TmpMin,
  aggmin(Rest,Min,X).
aggmin([X|Rest],Min,TmpMin) :-
  X >= TmpMin,
  aggmin(Rest,Min,TmpMin).

aggsum(List,Sum) :- aggsum(List,Sum,0).
aggsum([],X,X).
aggsum([H|T],Sum,Agg) :- Agg2 is Agg + H, aggsum(T,Sum,Agg2).

aggprod(List,Prod) :- aggprod(List,Prod,1).
aggprod([],X,X).
aggprod([H|T],Prod,Agg) :- Agg2 is Agg * H, aggprod(T,Prod,Agg2).
```

Bibliography

[1] M. Alviano and W. Faber. "Dynamic Magic Sets and super-coherent answer set programs". In: *AI Commun.* 24.2 (2011), pp. 125–145 (p. 89).

[2] R. Amadini, M. Gabbrielli, and J. Mauro. "An Empirical Evaluation of Portfolios Approaches for Solving CSPs". In: *CPAIOR*. Ed. by C. P. Gomes and M. Sellmann. Vol. 7874. Lecture Notes in Computer Science. Springer, 2013, pp. 316–324. ISBN: 978-3-642-38170-6. DOI: 10.1007/978-3-642-38171-3_21. URL: http://dx.doi.org/10.1007/978-3-642-38171-3_21 (p. 40).

[3] R. Amadini, M. Gabbrielli, and J. Mauro. "Features for Building CSP Portfolio Solvers". In: *CoRR* abs/1308.0227 (2013) (p. 40).

[4] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003 (pp. 30, 91).

[5] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. *ASP-Core-2 Input Language Format*. Tech. rep. ASP Standardization Working Group, 2013 (p. 25).

[6] R. A. Aziz, G. Chu, C. J. Muise, and P. J. Stuckey. "Stable Model Counting and Its Application in Probabilistic Logic Programming". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by B. Bonet and S. Koenig. AAAI Press, 2015, pp. 3468–3474. ISBN: 978-1-57735-698-1. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9709 (p. 115).

[7] R. A. Aziz, G. Chu, and P. J. Stuckey. "Stable model semantics for founded bounds". In: *TPLP* 13.4–5 (2013), pp. 517–532 (p. 21).

[8] M. Balduccini. "Industrial-Size Scheduling with ASP+CP". In: *LPNMR*. Ed. by J. P. Delgrande and W. Faber. Vol. 6645. Lecture Notes in Computer Science. Springer, 2011, pp. 284–296. ISBN: 978-3-642-20894-2. DOI: 10.1007/978-3-642-20895-9_33. URL: http://dx.doi.org/10.1007/978-3-642-20895-9_33 (p. 42).

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)". In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 1–15. ISBN: 0-89791-179-2. DOI: 10.1145/6012.15399. URL: http://doi.acm.org/10.1145/6012.15399 (p. 114).

[10] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 825–885. ISBN: 978-1-58603-929-5 (p. 91).

[11] C. Beeri and R. Ramakrishnan. "Special Issue: Database Logic Progamming On the power of magic". In: *The Journal of Logic Programming* 10.3 (1991), pp. 255–299. ISSN: 0743-1066. DOI: http://dx.doi.org/10.1016/0743-1066(91)90038-Q. URL: http://www.sciencedirect.com/science/article/pii/074310669190038Q (p. 114).

[12] A. Biere. "Adaptive Restart Strategies for Conflict Driven SAT Solvers". In: *SAT*. Ed. by H. Kleine Büning and X. Zhao. Vol. 4996. LNCS. Springer, 2008, pp. 28–33. ISBN: 978-3-540-79718-0 (p. 42).

[13] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5 (p. 40).

[14] H. Blockeel, B. Bogaerts, M. Bruynooghe, B. De Cat, S. De Pooter, M. Denecker, A. Labarre, J. Ramon, and S. Verwer. "Modeling Machine Learning and Data Mining Problems with FO(·)". In: *Proceedings of the 28th International Conference on Logic Programming - Technical Communications (ICLP'12)*. Ed. by A. Dovier and V. Santos Costa. Vol. 17. LIPIcs. Schloss Daghstuhl - Leibniz-Zentrum fuer Informatik, Sept. 2012, pp. 14–25. ISBN: 978-3-939897-43-9 (p. 30).

[15] B. Bogaerts, J. Jansen, B. De Cat, G. Janssens, M. Bruynooghe, and M. Denecker. "Bootstrapping inference in the IDP Knowledge Base System". In: *New Generation Computing* 34.3 (July 2016). URL: https://lirias.kuleuven.be/handle/123456789/519553 (pp. 64–66).

[16] M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. "Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3". In: *TPLP* 15 (6 Nov. 2015), pp. 783–817. ISSN: 1475-3081. DOI: 10.1017/S147106841400009X. URL: http://journals.cambridge.org/article_S147106841400009X (pp. 43, 118).

[17] K. L. Clark. “Negation as Failure”. In: *Logic and Data Bases*. Plenum Press, 1978, pp. 293–322. ISBN: 0-306-40060-X (p. 102).

[18] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. “GASP: Answer Set Programming with Lazy Grounding”. In: *Fundam. Inform.* 96.3 (2009), pp. 297–322 (p. 42).

[19] B. De Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, and M. Denecker. “Predicate Logic as a Modelling Language: The IDP System”. In: *CoRR* abs/1401.6312v2 (2016). URL: `http://arxiv.org/abs/1401.6312v2` (pp. 31, 109).

[20] B. De Cat, B. Bogaerts, and M. Denecker. *MiniSAT(ID) for satisfiability checking and constraint solving*. Ed. by A. Dovier and E. Pontelli. Sept. 2014. URL: `https://lirias.kuleuven.be/handle/123456789/463884` (p. 33).

[21] B. De Cat, B. Bogaerts, J. Devriendt, and M. Denecker. “Model Expansion in the Presence of Function Symbols Using Constraint Programming”. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*. IEEE Computer Society, 2013, pp. 1068–1075. ISBN: 978-1-4799-2971-9. DOI: `10.1109/ICTAI.2013.159`. URL: `http://dx.doi.org/10.1109/ICTAI.2013.159` (pp. 3, 33, 34, 42, 96, 109).

[22] B. De Cat, M. Denecker, M. Bruynooghe, and P. J. Stuckey. “Lazy Model Expansion: Interleaving Grounding with Search”. In: *J. Artif. Intell. Res. (JAIR)* 52 (2015), pp. 235–286. DOI: `10.1613/jair.4591`. URL: `http://dx.doi.org/10.1613/jair.4591` (pp. 42, 43, 96, 100, 112).

[23] B. De Cat, J. Jansen, and G. Janssens. “IDP3: Combining symbolic and ground reasoning for model generation”. In: *2nd Workshop on Grounding and Transformations for Theories With Variables*. 2013, pp. 17–24. URL: `https://lirias.kuleuven.be/handle/123456789/413766` (pp. 32, 64).

[24] K. Decroix, D. Butin, J. Jansen, and V. Naessens. “Inferring Accountability from Trust Perceptions”. In: *Information Systems Security, ICISS 2014, Hyderabad, 16-20 December 2014*. Ed. by A. Prakash and R. K. Shyamasundar. Springer-Verlag, Dec. 2014, pp. 69–88. URL: `https://lirias.kuleuven.be/handle/123456789/461505` (p. 118).

[25] K. Decroix, J. Lapon, B. De Decker, and V. Naessens. “A Formal Approach for Inspecting Privacy and Trust in Advanced Electronic Services”. In: *ESSoS*. Ed. by J. Jürjens, B. Livshits, and R. Scandariato. Vol. 7781. LNCS. Springer, 2013, pp. 155–170. ISBN: 978-3-642-36562-1 (p. 118).

[26] M. Denecker. "The Well-Founded Semantics Is the Principle of Inductive Definition". In: *JELIA*. Ed. by J. Dix, L. F. del Cerro, and U. Furbach. Vol. 1489. LNCS. Springer, 1998, pp. 1–16. ISBN: 3-540-65141-1 (pp. 25, 94).

[27] M. Denecker, G. Brewka, and H. Strass. "A Formal Theory of Justifications". In: *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*. Ed. by F. Calimeri, G. Ianni, and M. Truszczynski. Vol. 9345. Lecture Notes in Computer Science. Springer, 2015, pp. 250–264. ISBN: 978-3-319-23263-8. DOI: 10.1007/978-3-319-23264-5_22. URL: http://dx.doi.org/10.1007/978-3-319-23264-5_22 (pp. 25, 92).

[28] M. Denecker, M. Bruynooghe, and V. Marek. "Logic programming revisited: Logic programs as inductive definitions". In: *ACM Trans. Comput. Log.* 2.4 (2001), pp. 623–654 (p. 25).

[29] M. Denecker and D. De Schreye. "Justification Semantics: A Unifying Framework for the Semantics of Logic Programs". In: *Logic Programming and Nonmonotonic Reasoning, 2nd International Workshop, LPNMR 1993, Lisbon, Portugal, June 1993, Proceedings*. Ed. by L. M. Pereira and A. Nerode. MIT Press, 1993, pp. 365–379. ISBN: 0-262-66083-0. URL: https://lirias.kuleuven.be/handle/123456789/133075 (pp. 25, 92, 95, 96).

[30] M. Denecker, Y. Lierler, M. Truszczyński, and J. Vennekens. "A Tarskian Informal Semantics for Answer Set Programming". In: *ICLP (Technical Communications)*. Ed. by A. Dovier and V. S. Costa. Vol. 17. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 277–289. ISBN: 978-3-939897-43-9 (p. 25).

[31] M. Denecker and E. Ternovska. "A Logic of Nonmonotone Inductive Definitions". In: *ACM Trans. Comput. Log.* 9.2 (Apr. 2008), 14:1–14:52. ISSN: 1529-3785. URL: http://dx.doi.org/10.1145/1342991.1342998 (pp. 25, 33, 94).

[32] M. Denecker and J. Vennekens. "Building a Knowledge Base System for an Integration of Logic Programming and Classical Logic". In: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*. Ed. by M. García de la Banda and E. Pontelli. Vol. 5366. LNCS. Springer, 2008, pp. 71–76. ISBN: 978-3-540-89981-5. URL: http://dx.doi.org/10.1007/978-3-540-89982-2_12 (p. 27).

[33] M. Denecker and J. Vennekens. "The Well-Founded Semantics Is the Principle of Inductive Definition, Revisited". In: *KR*. Ed. by C. Baral, G. De Giacomo, and T. Eiter. AAAI Press, 2014, pp. 1–10. ISBN: 978-1-57735-657-8. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7957 (pp. 25, 94).

[34] M. Denecker and J. Vennekens. "Well-Founded Semantics and the Algebraic Theory of Non-monotone Inductive Definitions". In: *LPNMR*. Ed. by C. Baral, G. Brewka, and J. S. Schlipf. Vol. 4483. Lecture Notes in Computer Science. Springer, 2007, pp. 84–96. ISBN: 978-3-540-72199-4. DOI: 10.1007/978-3-540-72200-7_9. URL: http://dx.doi.org/10.1007/978-3-540-72200-7_9 (p. 93).

[35] M. Denecker, J. Vennekens, H. Vlaeminck, J. Wittocx, and M. Bruynooghe. "Answer Set Programming's Contributions to Classical Logic. An analysis of ASP methodology". In: *MG-65: Symposium on Constructive Mathematics in Computer Science*. Ed. by M. Balduccini and S. Tran. Lexington: Springer, Oct. 2010 (p. 25).

[36] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by E. Giunchiglia and A. Tacchella. Vol. 2919. LNCS. Springer, 2003, pp. 502–518. ISBN: 3-540-20851-8 (p. 31).

[37] EU Report. *Attainment: Raising graduate numbers*. http://ec.europa.eu/education/policy/higher-education/attainment_en.htm. Online; accessed 2016-09-01. 2016 (p. 1).

[38] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. "Using Database Optimization Techniques for Nonmonotonic Reasoning". In: *INAP Organizing Committee DDLP'99*. 1999, pp. 135–139 (p. 89).

[39] W. Faber, N. Leone, and S. Perri. "The Intelligent Grounder of DLV". In: *Correct Reasoning*. Ed. by E. Erdem, J. Lee, Y. Lierler, and D. Pearce. Vol. 7265. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 247–264. ISBN: 978-3-642-30742-3. URL: http://dx.doi.org/10.1007/978-3-642-30743-0_17 (pp. 88, 89, 114).

[40] Z. Fu, Y. Yu, and S. Malik. "Considering circuit observability don't cares in CNF satisfiability". In: *Design, Automation and Test in Europe*. Mar. 2005, 1108–1113 Vol. 2. DOI: 10.1109/DATE.2005.102 (p. 115).

[41] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. "Stream Reasoning with Answer Set Programming: Preliminary Report". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome,*

Italy, June 10-14, 2012. Ed. by G. Brewka, T. Eiter, and S. A. McIlraith. AAAI Press, 2012, pp. 613–617. ISBN: 978-1-57735-560-1 (p. 42).

[42] M. Gebser. "Proof theory and algorithms for answer set programming". PhD thesis. University of Potsdam, 2011. URL: http://opus.kobv.de/ubp/volltexte/2011/5542/ (p. 109).

[43] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. URL: dx.doi.org/10.2200/S00457ED1V01Y201211AIM019 (p. 30).

[44] M. Gebser, B. Kaufmann, and T. Schaub. "Conflict-driven answer set solving: From theory to practice". In: *Artif. Intell.* 187 (2012), pp. 52–89 (pp. 96, 102, 109).

[45] M. Gebser, T. Schaub, and S. Thiele. "GrinGo: A New Grounder for Answer Set Programming". In: *LPNMR*. Ed. by C. Baral, G. Brewka, and J. S. Schlipf. Vol. 4483. Lecture Notes in Computer Science. Springer, 2007, pp. 266–271. ISBN: 978-3-540-72199-4 (pp. 31, 88, 114).

[46] M. Gelfond and V. Lifschitz. "Classical Negation in Logic Programs and Disjunctive Databases". In: *New Generation Computing* 9.3/4 (1991), pp. 365–386 (p. 25).

[47] E. Goldberg and Y. Novikov. "BerkMin: A Fast and Robust SAT-Solver". In: *DATE*. IEEE Computer Society, 2002, pp. 142–149. ISBN: 0-7695-1471-5 (p. 41).

[48] Google. *Project Link*. https://www.google.com/get/projectlink/. Online; accessed 2016-09-01. 2016 (p. 1).

[49] P. M. Hill and J. W. Lloyd. *The Gödel programming language*. MIT Press, 1994, pp. I–XX, 1–348. ISBN: 978-0-262-08229-7 (p. 11).

[50] ISO Standards. *ISO/IEC Standard 14977 Download page*. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip. Online; accessed 2016-09-01. 2016 (p. 1).

[51] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. "Unfolding Partiality and Disjunctions in Stable Model Semantics". In: *ACM Trans. Comput. Logic* 7.1 (Jan. 2006), pp. 1–37. ISSN: 1529-3785. DOI: 10.1145/1119439.1119440. URL: http://doi.acm.org/10.1145/1119439.1119440 (p. 25).

[52] J. Jansen, B. Bogaerts, J. Devriendt, G. Janssens, and M. Denecker. "Relevance for SAT(ID)". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by S. Kambhampati. IJCAI/AAAI Press, 2016, pp. 596–603. ISBN: 978-1-57735-770-4. URL: http://www.ijcai.org/Abstract/16/091 (pp. 4, 91).

[53] J. Jansen, I. Dasseville, J. Devriendt, and G. Janssens. "Experimental Evaluation of a State-Of-The-Art Grounder". In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*. Ed. by O. Chitil, A. King, and O. Danvy. ACM, 2014, pp. 249–258. ISBN: 978-1-4503-2947-7. DOI: 10.1145/2643135.2643149. URL: http://doi.acm.org/10.1145/2643135.2643149 (pp. 4, 31, 45).

[54] J. Jansen, J. Devriendt, B. Bogaerts, G. Janssens, and M. Denecker. "Implementing a Relevance Tracker Module". In: vol. abs/1608.05609. 2016. URL: http://arxiv.org/abs/1608.05609 (pp. 4, 91).

[55] J. Jansen and G. Janssens. "Refining Definitions With Unknown Opens using XSB for IDP3". In: *CICLOPS*. Ed. by T. Ströder and T. Swift. Aachener Informatik Berichte AIB-2014-09. RWTH Aachen University, June 2014, pp. 15–29. URL: http://aib.informatik.rwth-aachen.de/2014/2014-09.pdf (pp. 4, 63, 84).

[56] J. Jansen, A. Jorissen, and G. Janssens. "Compiling Input∗ FO(·) Inductive Definitions into Tabled Prolog Rules for IDP3". In: *TPLP* 13.4–5 (2013), pp. 691–704. DOI: 10.1017/S1471068413000434 (pp. 4, 63).

[57] S. C. Kleene. "On Notation for Ordinal Numbers". English. In: *The Journal of Symbolic Logic* 3.4 (1938), pp. 150–155. ISSN: 00224812. URL: http://www.jstor.org/stable/2267778 (p. 93).

[58] T. L. Lakshman and U. S. Reddy. "Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System". In: *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*. Ed. by V. A. Saraswat and K. Ueda. MIT Press, 1991, pp. 202–217. ISBN: 0-262-69147-7 (p. 11).

[59] N. Leone, S. Perri, and F. Scarcello. "Improving ASP Instantiators by Join-Ordering Methods". In: *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '01. London, UK, UK: Springer-Verlag, 2001, pp. 280–294. ISBN: 3-540-42593-4. URL: http://dl.acm.org/citation.cfm?id=646400.688916 (p. 89).

[60] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. "The DLV system for knowledge representation and reasoning". In: *ACM Trans. Comput. Log.* 7.3 (2006), pp. 499–562 (pp. 31, 96).

[61] V. Lifschitz. "Answer set programming and plan generation". In: *Artificial Intelligence* 138.1 (2002), pp. 39–54 (p. 91).

[62] V. Marek and M. Truszczyński. "Stable Models and an Alternative Logic Programming Paradigm". In: *The Logic Programming Paradigm: A 25-Year Perspective*. Ed. by K. R. Apt, V. Marek, M. Truszczyński, and D. S. Warren. Springer-Verlag, 1999, pp. 375–398. URL: `http://arxiv.org/abs/cs.LO/9809032` (pp. 25, 91).

[63] M. Mariën. "Model Generation for ID-Logic". PhD thesis. Belgium: Department of Computer Science, KU Leuven, Feb. 2009 (pp. 93, 94).

[64] M. Mariën, J. Wittocx, and M. Denecker. "Integrating Inductive Definitions in SAT". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 2007, Proceedings*. Ed. by N. Derschowitz and A. Voronkov. Vol. 4790. LNCS. Springer, 2007, pp. 378–392. ISBN: 3-540-75558-6 (pp. 33, 91, 102, 109).

[65] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. "SAT(ID): Satisfiability of Propositional Logic Extended with Inductive Definitions". In: *SAT*. Ed. by H. Kleine Büning and X. Zhao. Vol. 4996. LNCS. Springer, 2008, pp. 211–224. ISBN: 978-3-540-79718-0 (pp. 3, 31, 40, 42, 91, 94, 96).

[66] J. P. Marques-Silva and K. A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521 (p. 91).

[67] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference*. ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2 (p. 41).

[68] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. "MiniZinc: Towards a Standard CP Modelling Language". In: *CP'07*. Ed. by C. Bessiere. Vol. 4741. LNCS. Springer, 2007, pp. 529–543 (p. 40).

[69] M. Ostrowski and T. Schaub. "ASP modulo CSP: The clingcon system". In: *TPLP* 12.4–5 (2012), pp. 485–503 (p. 42).

[70] L. Ryan. "Efficient Algorithms for Clause-Learning SAT Solvers". MA thesis. Vancouver, Canada: Simon Fraser University, 2004 (p. 41).

[71] P. J. Stuckey. "Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving". In: *CPAIOR*. 2010, pp. 5–9 (p. 91).

[72] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. "The TPTP Typed First-Order Form with Arithmetic." In: *LPAR*. Ed. by N. Bjørner and A. Voronkov. Vol. 7180. LNCS. Springer, 2012, pp. 406–419. ISBN: 978-3-642-28716-9 (p. 11).

[73] T. Swift, D. S. Warren, K. Sagonas, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, S. Dawson, and M. Kifer. *The XSB System Version 3.3.x Volume 1: Programmer's Manual*. 2013 (pp. 80, 81).

[74] T. Syrjänen. *Implementation of Local Grounding for Logic Programs with Stable Model Semantics*. Tech. rep. B18. Helsinki University of Technology, Finland, 1998 (pp. 31, 40, 88).

[75] A. Tarski and R. Vaugh. "Arithmetical extensions of relational systems". In: *Compositio Mathematica* 13 (1956), pp. 81–102 (p. 24).

[76] A. Tarski. "The semantic conception of truth and the foundations of semantics". In: *Philosophy and Phenomenological Research* 4 (1944) (p. 24).

[77] The World Bank. *World Development Report 2016: Digital Dividends*. The World Bank, 2016. DOI: 10.1596/978-1-4648-0671-1 (p. 1).

[78] G. S. Tseitin. "On the Complexity of Derivation in the Propositional Calculus, *Zapiski nauchnykh seminarov*". In: *LOMI* 8 (1968). English translation of this volume: Studies in Constructive Mathematics and Mathematical Logic, Part 2, A. O. Slisenko, eds. Consultants Bureau, N.Y., 1970, pp. 115-125, pp. 234–259 (pp. 32, 54, 74).

[79] U.S. Department of State. *Global Connect Initiative*. https://share.america.gov/globalconnect/. Online; accessed 2016-09-01. 2016 (p. 1).

[80] P. Vaezipoor, D. Mitchell, and M. Mariën. "Lifted Unit Propagation for Effective Grounding". In: *19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)* abs/1109.1317 (2011) (pp. 45, 46, 53–55, 88).

[81] A. Van Gelder, K. A. Ross, and J. S. Schlipf. "The Well-Founded Semantics for General Logic Programs". In: *J. ACM* 38.3 (1991), pp. 620–650. DOI: 10.1145/116825.116838. URL: http://dx.doi.org/10.1145/116825.116838 (pp. 25, 97).

[82] J. Verstichel. "The Lock Scheduling Problem (Het sluisplanningsprobleem)". De Causmaecker, Patrick and Vanden Berghe, Greet (supervisors). PhD thesis. Science, Engineering and Technology Group, Campus Kulak Kortrijk, Faculty of Science, Campus Kulak Kortrijk, Faculty of Engineering Science, Nov. 2013, p. 160. URL: https://lirias.kuleuven.be/handle/123456789/422310 (p. 112).

[83] H. Vlaeminck. "Applications of Feasible Inference for Expressive Logics". Approximating Definitions are introduced in chapter 3. PhD thesis. Leuven, Belgium: Department of Computer Science, K.U.Leuven, Dec. 2012 (p. 88).

[84] C. Williams and D. Strusani. *Value of Connectivity: Economic and social benefits of expanding internet access*. The Creative Studio at Deloitte, 2014 (p. 1).

[85] J. Wittocx. "Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic". PhD thesis. Leuven, Belgium: Department of Computer Science, K.U.Leuven, May 2010 (pp. 74, 88).

[86] J. Wittocx, M. Denecker, and M. Bruynooghe. “Constraint Propagation for Extended First-Order Logic”. In: *CoRR* abs/1008.2121 (2010) (p. 36).

[87] J. Wittocx, M. Denecker, and M. Bruynooghe. “Constraint Propagation for First-Order Logic and Inductive Definitions”. In: *ACM Trans. Comput. Logic* 14.3 (Aug. 2013), 17:1–17:45. ISSN: 1529-3785. URL: `http://doi.acm.org/10.1145/2499937.2499938` (pp. 33, 35).

[88] J. Wittocx, M. Mariën, and M. Denecker. “Grounding FO and FO(ID) with Bounds”. In: *J. Artif. Intell. Res. (JAIR)* 38 (2010), pp. 223–269 (pp. 33, 37).

Curriculum Vitae

Joachim Jansen was born on the 10th of October, 1989 in Turnhout, Belgium. After high school at the Sint-Pietersinstituut at Turnhout, he started with the studies of Informatics at the Katholieke Universiteit Leuven (KU Leuven) in 2007. In 2012 he received the degree of Master of Engineering in Computer Science (option Artificial Intelligence, cum laude).

In September 2012 he joined the DTAI (Declaratieve Talen en Artificiële Intelligentie) group of the Department of Computer Science at KU Leuven, to investigate additional techniques for systems supporting rich logics, under the supervision of prof. dr. ir. Gerda Janssens and with prof. dr. Marc Denecker as co-promotor.

List of Publications

A complete and up-to-date list of publications can be found at `http://www.cs.kuleuven.be/publicaties/lirias/mypubs.php?unum=U0085859`

Journal and Book Articles

- B. Bogaerts, J. Jansen, B. De Cat, G. Janssens, M. Bruynooghe and M. Denecker. "Bootstrapping inference in the IDP knowledge base system". In: New Generation Computing, volume 34, issue 3, pages 193-220.
- M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker and S. Verwer. "Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3". In: Theory and Practice of Logic Programming, volume 15, issue 6, pages 783-817.
- B. Bogaerts, B. De Cat, J. Jansen, M. Bruynooghe, B. De Cat, J. Vennekens and M. Denecker. "Simulating dynamic systems using linear time calculus theories". In: Theory and Practice of Logic Programming, volume 14, issue 4-5, pages 477-492, 2014.
- J. Jansen, G. Janssens, A. Jorissen. "Compiling input$*$FO$(\cdot)$ inductive definitions into tabled Prolog rules for IDP3". In: Theory and Practice of Logic Programming, volume 13, issue 4-5, pages 691-704, 2013.

Peer-reviewed Articles at Conferences and Workshops

- J. Jansen, B. Bogaerts, J. Devriendt, G. Janssens, M. Denecker. "Relevance for SAT(ID)". In: Twenty-Fifth International Joint Conference on Artificial Intelligence, 9-15th of July 2016, pages 596-603.
- J. Jansen, G. Janssens. "Translating ASP into a typed language without Herbrand functions". In: Workshop on Grounding, Transforming, and Modularizing Theories with Variables, 27th September 2015, pages 1-14.
- L. Lemaire, J. Vossaert, J. Jansen, V. Naessens. "Extracting Vulnerabilities in Industrial Control Systems using a Knowledge-Based System". In: International Symposium for ICS & SCADA Cyber Security Research, 17-18 September 2015, pages 1-10.
- B. Bogaerts, J. Jansen, B. De Cat, G. Janssens, M. Bruynooghe and M. Denecker. "Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development". In: International Workshop on Logic and Search, Vienna, July 18, 2014, pages 1-14.
- K. Decroix, D. Butin, J. Jansen, V. Naessens. "Inferring Accountability from Trust Perceptions". In: Information Systems Security, 16-20 December, 2014, pages 69-88.
- J. Jansen, I. Dasseville, J. Devriendt, G. Janssens. "Experimental evaluation of a state-of-the-art grounder". In: Principles and Practice of Declarative Programming, 8-10 September 2014, pages 249-259.
- J. Jansen, G. Janssens. "Refining definitions with unknown opens using XSB for IDP3". In: International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments, 17-18 July 2014, pages 15-29.
- B. De Cat, J. Jansen, G. Janssens. "IDP3: Combining symbolic and ground reasoning for model generation". In: Workshop on Grounding and Transformations for Theories with Variables, 12th International Conference on Logic Programming and Nonmonotonic Reasoning, 15 Sept 2013, pages 17-24.

Papers, Posters, and Presentations at Miscellaneous Events

- "Relevance for SAT(ID)", Twenty-Fifth International Joint Conference on Artificial Intelligence, 9-15th of July 2016.
- "Compiling input$*$FO($\cdot$) inductive definitions into tabled Prolog rules for IDP3", Flanders Scientific Research Community (WOG) Declarative Methods in Computer Science, 2013.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
KNOWLEDGE REPRESENTATION AND REASONING
Celestijnenlaan 200A box 2402
B-3001 Leuven
joachim.jansen@cs.kuleuven.be